NYO-1480-152
February 1970

Courant Institute of
Mathematical Sciences

AEC Computing and Applied Mathematics Center

The Algebraic Manipulation Package SYMBOLANG

A Manual for Users

Herbert J. Bernstein

AEC Research and Development Report

Mathematics and Computing
February 1970

New York University

UNCLASSIFIED

AEC Computing and Applied Mathematics Center
Courant Institute of Mathematical Sciences
New York University

Mathematics and Computing NYO-1480-152

The Algebraic Manipulation Package

SYMBOLANG

A Manual for Users

Herbert J. Bernstein

Contract No. AT[30-1]-1480

UNCLASSIFIED

PREFACE

This manual is intended to fill a long standing gap in the documentation of SYMBOLANG. Until now there has been no reasonably detailed reference for users of the package. Past reports have assumed much expertise on the part of the reader. We are probably guilty of the same sin, but hopefully to a lesser extent.

"Old hands" at SYMBOLANG will note that there have been major changes made. Functions of an arbitrary number of variables are now allowed. Expressions may appear as exponents, instead of exponents being restricted to constants. List space is conserved by allowing expressions to share common subexpressions. Evaluation and substitution have been combined and generalized. Most of the old routines are declared unfit for new code, and a bunch of strangers have taken their place. Yet, despite the brief flirtation with the name SYMB66, we still call the package SYMBOLANG.

Arnold Lapidus and Max Goldstein, the original developers of SYMBOLANG, provided an algebraic manipulation package with at least one major virtue - it could grow freely. The internal representation has accommodated itself well to extensions and could probably accept still more. Being a collection of subroutines for use by FORTRAN programs, the package can always absorb new features by simple addition of new routines. It is very likely that SYMBOLANG will continue to grow. At least the author hopes that to be the case.

The package itself is the work of Arnold Lapidus, Max Goldstein,

Susan S. Hoffberg, and, recently, the author. Great assistance has been provided by Samuel Greenspan and Alfred Magnus, and others who have been at the AEC Computing and Applied Mathematics Center of New York University. A singular debt is owed to Professor Goldstein for his inspiration and continued work on SYMBOLANG.

The author also wishes to thank Professor Nicholas Findler for providing a considerable part of the incentive for the writing of this manual, and Frances C. Bernstein for many hours of patient proofreading, indexing, and wifely cheer.

Herbert J. Bernstein

New York, New York

February 1970

CONTENTS

PREFACE ii

1. INTRODUCTION

1.1. An Overview 1.1

1.2. Background 1.3

1.3. Residual Notational Matters 1.11

1.4. Ordering and Simplification 1.12

1.5. System Access 1.15

1.6. Elementary Programming Techniques 1.19

1.7. Not so Elementary Programming Techniques 1.22

2. BASIC ROUTINES

2.1. Creation of Lists and Expressions 2.1

Exercises 2.4

2.2. Input of Expressions 2.5

Exercises 2.12

2.3. Output of Expressions 2.13

Exercise 2.16

2.4. Destruction of Lists and Expressions 2.17

Exercises 2.18

2.5. Other Output Facilities 2.19

2.6. Arithmetic Routines 2.21

Exercises 2.23

2.7. Definition and Evaluation 2.24

Exercises 2.27

2.8. Truncation 2.28

Exercises 2.30

2.9. Differentiation 2.31

Exercises 2.33

2.10. Analysis of Lists and Expressions 2.35

Exercises 2.41

2.11. Initialization 2.42

Exercises 2.47

2.12. Examples 2.49

3. BELLS, WHISTLES, AND FRILLS

3.1. Additional List and Expression Creation Methods 3.1

Exercises 3.12

3.2. Additional Expression Input Capabilities 3.14

3.3. Additional Expression Output Capabilities 3.18

Exercise 3.25

3.4. Niceties of List and Expression Destruction 3.26

3.5. Further Arithmetic Operations on Expressions 3.29

3.6. Definition and Evaluation Revisited 3.30

Exercise 3.46

3.7. Truncation Recapped 3.47

3.8. Further Means of Analysis of Lists and Expressions 3.51

3.9. Attribute-Value Lists 3.60

3.10. Another Example 3.62

4. REFERENCES

5. INDEX

1. INTRODUCTION

This is a user's manual for the algebraic manipulation package SYMBOLANG, a collection of FORTRAN callable subroutines to perform arithmetic operations, substitutions, evaluations, and differentiations on expressions represented as SLIP lists. This is not a text on algebraic manipulation, nor even on algebraic manipulation with SYMBOLANG, but a description of the external features and some internal features of version 3.34 of SYMBOLANG as implemented on a Control Data 6600 operating under the SCOPE 3.1.2 operating system using the RUN 2.3 FORTRAN compiler.

The reader is assumed to have access to and some familiarity with references [1],[2] and [14] which describe SCOPE 3.1.2, RUN 2.3 and SLIP respectively.

1.1. An Overview

The manual is divided into three major sections: INTRODUCTION which discusses background, system access, general structure, etc., BASIC ROUTINES which describes the most commonly used SYMBOLANG calls, and BELLS, WHISTLES, AND FRILLS which covers features which should not often be needed. Each section is further subdivided into subsections, some of which end with exercises. It is the author's intention that the user will at least once read the manual linearly rather than randomly, attempt the exercises, and ponder the examples. However, as a concession to those who can only read manuals randomly, an effort has been made to provide sufficient redundant information to permit considerable skipping. Thus the phrases "list or expression *lis*" and "variable or array element *var*" are semi-infinitely repeated

rather than rely on the reader's memory of notation.

The bulk of the information presented consists of descriptions of functions and subroutines. The user who feels a need for some sort of a preprocessor to facilitate his coding efforts should consult [9] for a description of the SYMBOLANG preprocessor QUEST, [11] for a simple general macro processor which might be useful for this purpose, or the example in §3.10 which happens to be a preprocessor for SYMBOLANG written in SYMBOLANG.

The attitude is assumed throughout that a function is a subroutine and that a subroutine of at least one argument may be treated as a function. Loev [10, pp. 52-53] provides a nice desciption of the results of such an attitude. Those reading this manual for the purpose of converting SYMBOLANG to another environment should be warned that many compilers consider this attitude subversive.

1.2. Background

SYMBOLANG was developed by Lapidus and Goldstein [8] in 1965 for use on an IBM 7094, extensively revised in 1967 by Lapidus, Goldstein, and Hoffberg [9] for use on a CDC 6600 under the Chippewa Operating System, and with major revisions and extensions brought to its current form in 1969 (see [3]). In its three incarnations it has been a package of FORTRAN callable subroutines, mostly written in FORTRAN and based on the list processing package SLIP developed by Weizenbaum [14], which allows one to perform such operations as multiplying the expression [[$1 + x$]] by the expression [[$1 - x$]] to obtain the expression [[$1 - x^2$]] rather than some numerical value. That is, SYMBOLANG is one of the large class of "formula manipulation" or "algebraic manipulation" systems such as are described in [5], [6], [12], and [13].

When dealing with numbers it is reasonable to allocate a certain fixed amount of space to represent each number, and just throw away any digits that overflow the allowed space as insignificant. With expressions, results may grow and shrink over a very wide range, and we rarely know what may be safely discarded. Rather than adopt the rather expensive approach of allowing the maximum possible space for each expression, it is common in formula manipulation to use list processing techniques which allocate to an expression just the space it needs and allow one to return unneeded space to a list of available space for future use.

The list processing system chosen for SYMBOLANG is SLIP, itself a package of FORTRAN callable subroutines mostly written in FORTRAN.

The SLIP routines allow one to create lists of items, which may themselves be lists, insert and delete items, and scan the items on a list. It is most convenient to think of a SLIP list as a parenthesized group of quantities:

$$(\ )$$

$$\text{or}\quad (quan_1\ quan_2\ \ldots\ quan_k)$$

where we call the first list empty, and the second list a list of k elements, the leftmost (top) element being the quantity $quan_1$ and the rightmost (bottom) element $quan_k$. In certain contexts we shall also consider the open and close parentheses as a special element of a list, the "head". Every list, even an empty list, has a head.

A SLIP list is referenced by a quantity which is interpreted by the routines of the package as a pointer to the head of the list, which itself contains pointers to the left- and rightmost elements of the list, which themselves contain pointers to the elements to their left and right, etc. We use the symbol *lis*, possibly with a subscript, to denote a quantity which we intend to be treated as a list within FORTRAN code. As we have done above, we use the symbol *quan*, for quantities on which we place no a priori restrictions. Such a quantity might be a floating point number, or an integer, or a list, or a Hollerith constant, etc.

When we wish a more restrictive notation, we use *int* for an integer quantity, *hol* for a Hollerith quantity, *symb* for a Hollerith

quantity without embedded blanks (called an "expression symbol"), *holarray* for a Hollerith quantity from which more than ten characters worth of information may be obtained, *var* for a variable or array element, *intvar* for an integer variable or array element, and *array* for an array. The user should refresh his memory of FORTRAN sufficiently to recall that a quantity may be a literal, variable, array element, or value of a function call, and that values of function calls may be used in places where variables or array elements are called for as arguments (causing the possible loss of that value).

Since a SLIP list may be placed on a SLIP list, certain precautions are needed to avoid losing track of lists. First, a list must never be placed on itself (see [15]). Second, quantities other than lists to be placed on a list are restricted to

floating point quantities in the range 10^{-293} to 10^{322} in magnitude

zero

integer quantities in the range $-(2^{30}-1)$ through $2^{30}-1$

Hollerith quantities (up to 10 characters)

We represent expressions as SLIP lists as follows:

$$\langle\text{expression}\rangle \leftarrow (\ \{\langle\text{term}\rangle\}_0^\infty\ )$$

$$\langle\text{term}\rangle \leftarrow (\ \mathit{quan}\ \{\langle\text{factor}\rangle\ \langle\text{power}\rangle\}_0^\infty\ )$$

$$\langle\text{factor}\rangle \leftarrow \{\langle\text{expression}\rangle\}_0^\infty\ \mathit{symb}$$

$$\langle\text{power}\rangle \leftarrow \mathit{quan}\ |\ \langle\text{expression}\rangle$$

where the left-hand side of each line represents the class of objects being defined, < and > enclose class names, { and } indicate that the enclosed objects are to be included in the definition for each number of repetitions in the range indicated by the numbers on the right brace, | separates alternative definitions, *quan* is treated as the class of floating point quantities other than zero, and *symb* is the class of expression symbols. The meaning of this syntax is simple: an expression is a list of terms and represents the sum of those terms; a term is a list beginning with a constant coefficient followed by factors with powers and represents the product of the constant by the factors raised to the powers; a factor is a list fragment consisting of expressions followed by an expression symbol and represents the expression symbol applied as a function to the expressions as its arguments or represents the expression symbol itself if there are no arguments; and a power is either an expression or a constant.

For convenience in displaying examples of expressions as lists, we introduce commas between list elements, and use double brackets as above to enclose real world expressions.

The expression [[0]] may be thought of as a sum of no terms. We represent it as the list ( ) which is empty.

The expression [[36.5]] is another example of a "constant" expression; i.e. it involves only numbers,not symbols. It consists of a single term which has only a coefficient. We represent it as

the list ((36.5)) which has one element, a sublist. The sublist has one element, the number 36.5.

The expression [[x]] is a non-constant expression consisting of one term with coefficient one, a single factor 1HX, and the power one. We represent it as the list ((1., 1HX, 1.)).

The expression [[$-18 sin(x)^y z^3 + 1$]] is a moderately complex expression, which we represent as the list ((1.), (-18., 1HZ, 3., ((1., 1HX, 1.)), 3HSIN, ((1., 1HY, 1.)))).

The expression [[$user(0, x)$]] involves a function of more than one variable. We represent it as the list ((1., ( ), ((1., 1HX, 1.)), 4HUSER, 1.)).

In general, we choose a Hollerith constant of from 1 to 10 non-blank characters to represent each symbol in the expression, represent all function arguments as lists, break up the expression into a sum of terms, and each term into a product of factors raised to powers, and introduce the trivial coefficient and power one where necessary. All numbers are in floating point, and usually one uses FORTRAN variable naming conventions for forming expression symbols. Rather than force ourselves to use the distributive law to expand parenthesized subexpressions, which may not even be possible with fractional powers, we treat parentheses without a function name as an appearance of a special function name which we arbitrarily translate into the expression symbol 3H1.*.

In this manner we may represent expressions as general as

$$\sum_{i=0}^{m} c_i \prod_{j(i,0)=0}^{m(i,0)} v_{0,j(i,0)}^{p_{0,j(i,0)}} \prod_{j(i,1)=0}^{m(i,1)} v_{i,j(i,1)}(e_{1,j(i,1)})^{p_{1,j(i,1)}}$$

$$\cdot \prod_{j'(i,1)=0}^{m'(i,1)} (e'_{i,j(i,1)})^{p'_{i,j(i,1)}}$$

$$\cdot \prod_{j(i,2)=0}^{m(i,2)} v_{2,j(i,2)}(e_{2,j(i,2),1},\ e_{2,j(i,2),2})^{p_{2,j(i,2)}}$$

$$\vdots$$

$$\cdot \prod_{j(i,r_i)=0}^{m(i,r_i)} v_{r_i,j(i,r_i)}(e_{r_i,j(i,r_i),1},\ \ldots$$

$$\ldots,\ e_{r_i,j(i,r_i),r_i})^{p_{r_i,j(i,r_i)}}$$

as the following list:

$$\Bigg\{\bigg\{\Big[c_i\{v_{0,j(i,0)}\ p_{0,j(i,0)}\}_{j(i,0)=0}^{m(i,0)}$$

$$\{e_{1,j(i,1)}\ v_{1,j(i,1)}\ p_{1,j(i,1)}\}_{j(i,1)=0}^{m(i,1)}$$

$$\{e'_{1,j(i,1)}\ \text{3H1.*}\ p'_{i,j(i,1)}\}_{j'(i,1)=0}^{m'(i,1)}$$

$$\{e_{2,j(i,2),1}\ e_{2,j(i,2),2}\ v_{2,j(i,2)}\ p_{2,j(i,2)}\}_{j(i,2)=0}^{m(i,2)}$$

$$\vdots$$

$$\{e_{r_i,j(i,r_i),1}\ ,\ldots$$

$$\ldots\ e_{r_i,j(i,r_i),r_i}\ v_{r_i,j(i,r_i)}\ p_{r_i,j(i,r_i)}\}_{j(i,r_i)=0}^{m(i,r_i)}\ \Big]\bigg\}_{i=0}^{m}\Bigg\}$$

What has been presented so far does not force any degree of uniqueness in the representation of an expression as a list. That matter is handled in §1.4. For most purposes the user need not concern himself with this matter. Very complex expressions may be formed by use of the routine INLIST described in §2.2 and §3.2. Reasonably complex expressions may be built "by hand" from simple expressions of just one term with at most one factor by addition, multiplication, and taking powers using the arithmetic routines in §2.6 and the list construction routines in §2.1.

1.3. Residual Notational Matters

We are about to look at some actual functions, so notation had best be made clear.

An expression within SYMBOLANG code is, of course, just a particular form of a list. We use the symbol *lis* to represent an expression as well as any other list. However, there are many routines which will hang or do other interesting things if presented with an arbitrary list where they expect an expression. Further, there are many list modifications which are perfectly proper for lists other than expressions, but which would cause too much disorganization if applied to expressions. Nondestructive operations specified for lists may always be applied to expressions. Thus the user should be careful to distinguish the phrases "list *lis*", "expression *lis*", and "list or expression *lis*".

In most cases where a Hollerith quantity, *hol* or *symb*, is called for, only the first ten characters will be used. Also, by Hollerith, we mean exactly that, nnH, not nnL or nnR.

Where the difference is important, the letter O will be distinguished from the digit 0 by writing the letter as Ø. Thus we use SYMBØLANG or SYMBOLANG, but not SYMB0LANG.

Some functions which return lists or expressions have floating point names. Most have integer names. The user is cautioned to avoid undesired mode conversion.

1.4. Ordering and Simplification

All right thinking users will agree that A*B*C-C*B*A is best treated as zero. Not everyone will agree that X-(X) is best treated as zero. SYMBOLANG is right thinking but not everyone. Identical terms differing only in their coefficients are combined, and a zero coefficient causes a term to be dropped from an expression. Identical factors differing only in their powers are combined, and a zero power causes a factor to be dropped. If the constant expressions [[0]] and [[1]] are raised to any power, they are unchanged. In short, when operating on expressions, SYMBOLANG performs complete identity removal in the sense of [13, p. 557]. In order to keep this task finite, SYMBOLANG imposes an ordering on all expressions and their components. All operations on expressions preserve this ordering, some assume it, and most impose it.

The user may simplify a hand created expression by calling the routine SMPL.

SMPL(*lis*) reorganizes the expression *lis* into simplified form, returning *lis* as its value. This is an actual change to the expression, and one of the few cases where it is proper to alter an existing expression.

This operation is rather expensive, and should be avoided as unnecessary in most cases, since simplification spreads in SYMBOLANG with lightning speed.

The ordering imposed is built up from orderings on numbers and expression symbols. Numbers are ordered arithmetically, and expression symbols lexicographically using the following ordered alphabet:

0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F	G
H	I	J	K	L	M	N	Ø	P	Q	R	S	T	U	V	W	X
Y	Z	+	-	*	/	(	)	$	=	,	.	≡	[	]	:	≠
→	∨	∧	↑	↓	<	>	≤	≥	¬							

with blank coming before all other characters. Thus 3H1.* is less than 1HA, which is less than 2HAA, which is less than 1H$.

We now build a lexicographic ordering of expressions from the following rules:

A. All numbers are less than all expressions.

B. The empty expression ( ) is less than all non-empty expressions.

C. An expression ($term_1$, $term_2$, ..., $term_k$) is less than the expression ($term'_1$, $term'_2$, ..., $term'_{k'}$) if

(1) $term_1$ is less than $term'_1$; or

(2) there is an i such that $term_i$ exists and is less than $term'_i$ which also exists, and $term_j = term'_j$ for $j = 1, 2, \ldots, i-1$; or

(3) k is less than k' and $term_j = term'_j$ for $j = 1, \ldots, k$.

D. A term with just a coefficient is less than a term with factors, and also less than any other term with just a coefficient where that coefficient is greater.

E. A term (*coe*, $factor_1$, $power_1$, ..., $factor_k$, $power_k$) is less than the term (*coe'*, $factor'_1$, $power'_1$, ..., $factor'_{k'}$, $power'_{k'}$) if

(1) $factor_1$ is less than $factor'_1$; or

(2) there is a leftmost factor or power in the first term which is less than the same numbered factor or power in the second term, and where all factors and powers to the left agree; or

(3) k is less than k' and all factors and powers through the kth agree; or

(4) the terms differ only in their coefficients, and the first term has a lesser coefficient.

F. A factor without arguments is less than any factor with arguments, which in turn is less than any factor with a still greater number of arguments.

G. Two factors with an equal number of arguments are compared first by their expression symbols. If those are equal, then the arguments are compared from left to right.

Factors within terms and terms within expressions are reordered to achieve expressions of the lowest possible ordering. Constant expression powers are replaced by the constants, terms and factors are combined where possible, and dropped when trivial. Parentheses are not handled any differently from any other function.

The result is to produce expressions in which constant terms lead, symbols without arguments come next, then functions of one variable, two, etc.

1.5. System Access

The user gains access to SYMBOLANG by running ordinary batch production jobs. It may also be possible to use SYMBOLANG in a batch-like time sharing environment as in [7], but the package is not interactive in the sense of [6].

Assuming normal installation procedure has been followed, some of the routines will have been added to the system library, and the rest will be obtained by use of the programs SLIP [3] and SELECT [4] which should themselves be in the system library. In such an environment, a SYMBOLANG job would have the structure

```
  job card
  RUN(S)        CØMPILE USER PRØGRAM TØ FILE LGØ
  SLIP(LIB)     PUT SLIP AND SYMBØLANG LIB ØN LIB
  SELECT(N)     TRANSFER NEEDED RØUTINES FROM LIB TØ LGØ
  LGØ.          LØAD AND EXECUTE LGØ
end-of-record card
  user program and subroutines
end-of-record card
  user data
end-of-file card
```

The job card should allow a field length of at least 70000 octal words. Time depends completely on the task. The user program and subroutines should follow ordinary FORTRAN coding conventions, subject to the following constraints:

The program should provide the standard input unit, 5LINPUT, and the standard output unit, 6LOUTPUT, by starting with a program header card, say, of the form

```
PRØGRAM TEST(INPUT,ØUTPUT)
```

A list of available space must be initialized, see §2.11, say by

```
DIMENSIØN SPACE(5000)
CALL INITAS(SPACE,5000)
```

This call to INITAS also provides 100 "public" lists in blank CØMMØN which may be used for scratch work. They are accessed by

```
CØMMØN AVSL,X(100)
```

where AVSL is a pointer not to be tampered with, and X is the array of public lists. X(99) and X(100) are used extensively within SYMBOLANG and should not be touched.

CØMMØN blocks with labels consisting of six characters and beginning with LSQ should not be created by the user, nor should he create one with the label SYMSYS.

In addition to entry points described in this manual, the user should not create any routines with names of six characters beginning with LSQ, LTQ, XSQ, or XTQ.

The generality of the last constraint is to allow for the future expansion of the package. The entry points currently in the package are:

ADD	ADVLEL	ADVLER	ADVLL	ADVLNL
ADVLNR	ADVLR	ADVLWL	ADVLWR	ADVSEL
ADVSER	ADVSL	ADVSNL	ADVSNR	ADVSR
ADVSWL	ADVSWR	ALØAD	BØT	BREAK
CPYTRM	DELETE	DVSUM	DVTRM	EVALUE
GETCØE	HITENT	INITAS	INITRD	INLIST
INLSTL	INLSTR	INSBST	INSUBT	INTARG
INTENT	IRALST	IRARDR	ITSVAL	KØKE
LCNTR	LDATVL	LDIF	LEXICØ	LIST
LISTAV	LISTMT	LISTØN	LØCARG	LØCT
LØFRDR	LØØKUP	LPNTR	LRDRCP	LRDRØV
LSQADD	LSQCDR	LSQCMP	LSQCNM	LSQCPY
LSQCXP	LSQDEF	LSQDES	LSQDSF	LSQERR
LSQGAR	LSQGNF	LSQIDR	LSQINI	LSQLCØ
LSQMEX	LSQMNL	LSQMTM	LSQNAM	LSQØUT
LSQPNT	LSQRAZ	LSQSBS	LSQTRC	LSQTYP
LSQUNM	LSQVAL	LSQVVL	LSSCPY	LSTEQL
LVLRV1	LVLRVT	MADATR	MADLFT	MADNBT
MADNTP	MADRGT	MAKEDL	MANY	MØNØ
MRKLSS	MRKLST	MTDLST	MTLIST	NAMEDL
NAMTST	NEWBØT	NEWTØP	NEWVAL	NØATVL
NUCELL	NULSTL	NULSTR	NUMPY	NXTLFT
NXTRGT	PØPBØT	PØPMID	PØPTØP	PØWER
PRIPUT	PRLSTS	PUTLST	RCELL	REED
SAME	SBST	SEQLL	SEQLR	SEQRDR
SEQSL	SEQSR	SMPL	SØLVE	SRTRM
SUB	SUBSBT	SUBST	SUBSTP	SUBT

SUMPY	TERM	TØP	TRCAL	TRUNC
TSTCØN	VALARG	VISIT		

The following entry points are usually obtained from the system library and should not be redefined:

ABNØRML	ACGØER	ADVIN.	ALØG	ALØG10
AND	ATAN	BACCHK=	BKSPRU.	CIØ1.
CØNT	CØS	DAT.	END	EQUAL
EXIT	EXP	FIZBAK.	FIZBA.	GETBA
KØDER	ID	IFENDF	IFTHEN	INCHEK
IND	INHALT	INIBIP	INPUTC	INPUTS
INTGER	KRAKER	LANØRM	LNKL	LNKR
MADØV	MVWDS.	ØPEN.	ØR	ØUTPTC
ØUTPTS	ØVWRT	PØSFIL.	PØSFI.	Q8NTRY
RBAIEX	RBAREX	RDPRU.	REVIND	SBARGS
SETDIR	SETIND	SETRAY	SHIFTR	SHIN
SIN	SIØ.	SQIN	SQØUT	STØP
STRDIR	STRIND	SYSTEM	SYSTEMC	SYSTEMP
TAN	THENIF	ZERØ		

This list is a mixture of SLIP, SYMBOLANG, and miscellaneous routines, not all of which will be described. Unless there is some pressing reason to do otherwise, we shall refer to all as SYMBOLANG routines on the theory that SYMBOLANG is an extension of SLIP which is an extension of FORTRAN.

1.6. Elementary Programming Techniques

The most commonly repeated pattern of code in SYMBOLANG programming is

create operands

perform expression operation

erase unneeded operands

For example, using the input techniques of §2.2, addition as per §2.6 and erasure as per §2.4, we may input,add and erase two expressions by

```
LA=INLIST(LA,5LINPUT)
LB=INLIST(LB,5LINPUT)
LC=LSQADD(LA,LB)
CALL LSQDES(LA)
CALL LSQDES(LB)
```

The erasure of unneeded operands is very important to avoid running out of space from which to create expressions.

Since mode conversions are often not desired, information may be transferred without conversion by

STRDIR(*quan*, *var*) which stores the quantity *quan* into the variable or array element *var* without mode conversion.

INTGER(*quan*) provides the quantity *quan* with an integer name.

MØNØ(*quan*) also provides the quantity *quan* with an integer name.

REAL(*quan*) provides the quantity *quan* with a floating point name.

SAME(*quan*) also provides the quantity *quan* with a floating point name.

The final programming technique to be covered is recursion. The routines TERM and VISIT provide the ability to recursively execute blocks of FORTRAN code. The entry to the code is provided by executing an ASSIGN statement to store the location of the first line of code to be executed in an integer variable. Since the form of an ASSIGN statement is usually

```
ASSIGN 100 TØ LØC
```

a variable established this way will be designated by the symbol *loc*.

VISIT(*loc*) saves the location from which the call is made and transfers control to the FORTRAN statement at the location specified by *loc*. VISIT return a value which is provided as the argument to the call to TERM which returns control.

TERM(*quan*) returns control to the location from which the last VISIT was paid, discarding that location from the list on which it was saved. The quantity *quan* is returned as the value

of VISIT.

For example, the following subroutine will compute the function N*(N-1)*(N-2)*...*2*1 recursively.

```
      FUNCTIØN IFAC(N)
      ASSIGN 100 TØ LØC
      M=N
      CALL STRDIR(VISIT(LØC),IFAC)
      RETURN
100   IF(M.LE.1)CALL TERM(1)
      M=M-1
      CALL STRDIR(VISIT(LØC),K)
      M=M+1
      CALL TERM(M*K)
      END
```

Note that we have been careful to restore M to its original state on each pass through the code. More complex saving and restoring may be done with the routines MANY and LSQUNM described in §2.1 and §2.4.

1.7. Not so Elementary Programming Techniques

SYMBOLANG routines make use of some special facilities with which the user may wish to enhance his own code.

The short simple routines such as STRDIR, which fit into one word of machine code, actuallv replace the code used to call them with in-line code. This is done when the routine is executed, so that a loop of such calls will execute the routine once and the in-line code thereafter. (This execution time overwriting is believed to be unique to SYMBOLANG). The effect on most SYMBOLANG code is an improvement of 30% over the timing with more conventional coding. If the user has a similar short routine, he may make it overwrite its call with the routine ØVWRT.

ØVWRT(*quan*) replaces the call to the routine which called ØVWRT with the word of machine code given by the quantity *quan*. For example ØVWRT(46000460004600046000B) would replace the call with a do-nothing instruction.

Some compression of code can be achieved by use of the functions IFTHEN and THENIF which act as conditional expressions.

IFTHEN($quan_1$, $quan_2$, ..., $quan_k$) for k = 1, 2, 3, ..., 60, considers its odd numbered arguments (except possibly the last) to be logical expressions, and returns the leftmost even numbered

argument following an odd numbered argument which is TRUE. If the total number of arguments is odd, the last argument is returned if all the other odd numbered arguments are FALSE. If the total number of arguments is even, and no odd numbered argument is TRUE, zero is returned.

THENIF is identical to IFTHEN, except for the name.

For example

```
X=THENIF(J.LT.0,0.,J.GT.5,5.,FLØAT(J))
```

is a short way of writing

```
X = FLØAT(J)
IF(J.LT.0)X=0.
IF(J.GT.5)X=5.
```

IFTHEN and THENIF are examples of SYMBOLANG routines which handle variable length argument lists. The user may write routines which handle such argument lists by use of the functions SBARGS, LØCARG, INTARG, and VALARG, as well as the subroutine ALØAD.

SBARGS(*intvar*) stores the number of actual arguments used in the call to the routine which called SBARGS in the integer variable or array element *intvar*. The number of arguments is also

returned as the function value.

ALØAD(*array*, var_1, var_2, ..., var_k) for $k = 1, 2, \ldots, 59$, stores the machine addresses of the variables var_i into the array elements *array*(i) where *array* is an array of dimension k. Such a call to ALØAD should be made to store the addresses of the first six arguments of any routine which wishes to use LØCARG, INTARG, or VALARG.

LØCARG(*int*) returns the machine address of argument *int* of the routine which called LØCARG. The integer quantity *int* must lie between 1 and the actual number of aguments used, and a call to ALØAD for the first six arguments must have been made.

INTARG(*int*) returns the value of argument *int* of the routine which called INTARG. The same restrictions apply as do to LØCARG.

VALARG is INTARG with another name.

Thus a subroutine which wishes to index through its argument list might begin

```
FUNCTIØN GØGØ(A,B,C,D,E,F,G,H,I,J,K,L,M,N,Ø)
DIMENSIØN LARG(6)
CALL SBARGS(NARGS)
CALL ALØAD(LARG,A,B,C,D,E,F)
```

```
DØ 100 JJ=1,NARGS
      .
      .
  . . . = VALARG(JJ) . . .
      .
      .
```

Since machine addresses have been mentioned, we will look at a few routines to make some use of them. In the rest of this manual the concept of machine address will be avoided assiduously.

STRIND(*quan*, *int*) stores the quantity *quan* into the machine address specified by the integer quantity *int*.

MADØV(*var*) returns the machine address of the variable or array element *var*. Thus STRIND(*var*, MADØV(*var*)) is a peculiar way to do nothing.

CØNT(*int*) returns the contents of machine address *int*.

INHALT(*int*) also returns the contents of machine address *int*, but with an integer name.

These last four routines are examples of SLIP "primitives". The full set consists of CØNT, ID, INHALT, LNKL, LNKR, MADØV, SETDIR, SETIND, STRDIR, STRIND, AND, EQUAL, INTGER, LANØRM, ØR, SHIN,

SQIN, and SQØUT. See [10] and [14] for details on these routines.

2. BASIC ROUTINES

The following are the more important calls to SYMBOLANG routines. Since the full package consists of over 150 routines, some with several alternative calling sequences, the decision as to which are "basic" is somewhat arbitrary.

2.1. Creation of Lists and Expressions

The functions LIST, LSQMNL, MANY, and LSSCPY may be used to construct lists and expressions from scratch or from existing lists.

LIST(9) returns a new empty list, ( ), which is the correct representation of [[0]].

LSQMNL($quan_1$, $quan_2$, ..., $quan_k$), k = 1, 2, ..., 60, returns a new list containing the quantities in the argument list, i.e. ($quan_1$, $quan_2$, ..., $quan_k$). For example, the following return the correct representations of the indicated expressions:

[[3.6]]	LSQMNL(LSQMNL(3.6))
[[1]]	LSQMNL(LSQMNL(1.))
[[-6.9E36]]	LSQMNL(LSQMNL(-6.9E36))
[[x]]	LSQMNL(LSQMNL(1.,1HX,1.))
[[$5y^3$]]	LSQMNL(LSQMNL(5.,1HY,3.))
[[$-4.5 sin(alpha)^{beta}$]]	LSQMNL(LSQMNL(-4.5, LSQMNL(LSQMNL(1.,5HALPHA,1.)),3HSIN, LSQMNL(LSQMNL(1.,4HBETA,1.)))

[[$user_1(0,5)$]] LSQMNL(LSQMNL(1.,LIST(9),LSQMNL(LSQMNL(5.)),
5HUSER1,1.))

MANY(*lis*, $quan_1$, $quan_2$, ..., $quan_k$), k = 1, 2, ..., 59, appends $quan_1$, $quan_2$, ..., $quan_k$, in turn, to the bottom of *lis*, returning *lis* as its value. This is an actual change to the list.

```
LA = LSQMNL(LQ1,LQ2,LQ3,LQ4)
```

is equivalent to

```
LA = LIST(9)
CALL MANY(LA,LQ1,LQ2,LQ3,LQ4)
```

and to

```
LA = LSQMNL(LQ1)
CALL MANY(LA,LQ2,LQ3,LQ4)
```

This routine is especially useful for stacking items to be saved during recursion. It may also be used to build up expressions. For example, the representation of [[$1+a(0,x^3,7)$]] may be constructed by:

```
C   PUT FIRST TERM IN AN EXPRESSIØN
       LAT1 = LSQMNL(LSQMNL(1.))
C   BEGIN THE SECØND TERM
       LATT2 = LSQMNL(1.)
C   FØRM THE ARGUMENTS FOR USE BY THE SECØND TERM
       LAR1 = LIST(9)
       LAR2 = LSQMNL(LSQMNL(1.,1HX,3.))
       LAR3 = LSQMNL(LSQMNL(7.))
C   CØMPLETE THE SECØND TERM
```

```
      CALL MANY(LATT2,LAR1,LAR2,LAR3,1HG,1.)
C   FØRM THE FINAL EXPRESSIØN
      CALL MANY(LAT1,LATT2)
```

(This last step is correct only because the ordering of terms is clear here. In general, LSQADD as described in §2.6 should be used.)

LSSCPY(*lis*) returns a complete copy of the list or expression *lis*. This is useful when a routine destroys or modifies an argument which must be available later.

Exercises

2.1.1. Write code to generate the correct representation of the following expressions:

[[0]]	[[1/3]]	[[$x^3/3$]]	[[$x(y)/3$]]
[[1]]	[[x]]	[[$-x^3/3$]]	[[$x(3y)$]]
[[-1]]	[[$x/3$]]	[[$x^y/3$]]	[[$-2x(3y)^{-x}$]]

2.1.2. Describe the list LB that results from the following code:

```
      LA=LSQMNL(1.)
      LAA=LSQMNL(-2.5,LSQMNL(LA),LSQMNL(LSSCPY(LA)),
     *   1HX,LA)
      LB=LIST(9)
      CALL MANY(LA,1HY,1.)
      CALL MANY(LB,LAA)
```

2.1.3. What expression does the list LB formed above represent?

2.1.4. The list ((1.,1HX,-1.,1HX,1.)) is not a proper expression since the reduction to ((1.)) by cancelling the powers of 1HX, is required. Use the expression symbol 3H1.* as the identity function to represent parenthesization, and write code for the expression [[$x/(x)$]]. (Do not expend too much energy worrying about proper ordering.)

2.2. Input of Expressions

The function INLIST provides a means of translating FORTRAN-like expressions into their internal representation on lists. Input may be taken from a file or from line images saved on a list.

INLIST(*var*, *quan*) returns a list representing the expression found on the logical unit or list *quan*. The variable or array element *var* will also contain the newly created expression. Regardless of whether *quan* is a unit number or a list, input is handled line by line, scanning for an expression in columns 7 through 72 of each line until either a $ is encountered or a line is encountered in which any of columns 1 through 5 are non-blank. The end of the input stream will also terminate the scan. The following syntax is applied to the characters obtained in this manner after removing any blanks.

$$\langle\text{expression}\rangle \leftarrow \langle\text{term}\rangle \; \{\langle\text{addition operator}\rangle\langle\text{term}\rangle\}_0^{\infty}$$

$$\langle\text{term}\rangle \leftarrow \langle\text{factor}\rangle \; \{{*}{*}\langle\text{factor}\rangle\}_0^{\infty} \; \{\langle\text{multiplication operator}\rangle \; \langle\text{factor}\rangle \; \{{*}{*}\langle\text{factor}\rangle\}_0^{\infty}\}_0^{\infty}$$

$$\langle\text{factor}\rangle \leftarrow \{\langle\text{addition operator}\rangle\}_0^{\infty} \; \{\langle\text{number}\rangle \mid \langle\text{symbol}\rangle \mid \{\langle\text{symbol}\rangle\}_0^{1}(\; \langle\text{expression}\rangle \; \{ \; , \; \langle\text{expression}\rangle\}_0^{\infty} \;)\}$$

$$\langle\text{addition operator}\rangle \leftarrow + \mid -$$

$$\langle\text{multiplication operator}\rangle \leftarrow * \mid /$$

$$\langle\text{symbol}\rangle \leftarrow \langle\text{letter}\rangle \; \{\langle\text{letter}\rangle \mid \langle\text{digit}\rangle \mid .\}_0^{9}$$

$$\langle\text{number}\rangle \leftarrow \langle\text{whole}\rangle \; \{\langle\text{fraction}\rangle\}_0^{1} \; \{\langle\text{exponent}\rangle\}_0^{1} \mid \langle\text{fraction}\rangle \; \{\langle\text{exponent}\rangle\}_0^{1}$$

$$\langle\text{whole}\rangle \leftarrow \{\langle\text{digit}\rangle\}_1^{\infty}$$

<fraction> ← . {<digit>}$_0^\infty$

<exponent> ← E {<addition operator>}$_0^\infty$ <whole>

<letter> ← A | B | C | D | E | F | G | H | I | J | K | L |
M | N | Ø | P | Q | R | S | T | U | V | W | X |
Y | Z

<digit> ← 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9

Thus expressions are formed from symbols and numbers combined by the arithmetic operators +, -, *, /, and **, balanced parentheses and commas.

Symbols are strings of from 1 to 10 letters, digits, or periods, which begin with a letter. For example, the following are valid symbols.

A	A..1	LABØRPARTY
X.1.2.ABC	ETC...	DIMENSIØN
SIN	CØS	TAN
FUNCTIØN	PRØGRAM	LSQ249

The following are not valid symbols.

.	1..A	LABØRPARTIES
$A=	SIN(X)	.NØT.
.TRUE.	1E10	-B
X+1	*A	1.*

Numbers are combinations of a decimal whole number with a decimal fraction and exponent. Either the whole number or the fraction must appear, but in either case the remaining two fields are optional. If all fields are present a number assumes the form

$n.m \text{ E } k$

which is interpreted as

$(n.m)\ 10^k$

while, if some fields are omitted, a number may have the forms

$n \qquad n.m \qquad n \text{ E } k \qquad .m \qquad .m \text{ E } k$

which are interpreted as

$n. \qquad n.m \qquad (n.)\ 10^k \quad 0.m \qquad (0.m)\ 10^k$

where n is a string of one or more digits, m is either empty or a string of digits, and k is a string of one or more digits optionally preceded by a string of pluses and minuses determining the sign of the exponent. The following are examples of valid numbers with the indicated interpretations in parentheses:

.	(0.0)	1	(1.0)
5E-1	(0.5)	.5	(0.5)
1.E3	(1000.0)	1.E--3	(1000.0)
1.1E-2	(0.0011)	1.1E+2	(110.0)
0	(0.0)	0E10	(0.0)

Symbols are stored in the resulting expression directly as expression symbols; i.e., the characters of the symbols are used left justified with blank fill. Numbers are stored as expression numbers; i.e., the number is translated into a single precision floating point number, holding between 14 and 15 digits accuracy with magnitudes ranging from approximately 10^{-294} to 10^{322} for non-zero numbers.

A symbol may be used as a function name by following it with an argument list of expressions separated by commas and enclosed

in parentheses. If the argument list is present without a preceding function name, the expression symbol 3H1.* is used as the function name in the resulting expression. For example, the following are valid function references:

```
(X + 1.1)            SIN(3.14*Y.2)
(0,(((2**3))),K)     TAN(X**(EXP,SIN),+-+.,A/B)
(((CØS(0))))         CØS((((0))))
A.......(3/B.....)   X(X(X,X),X)
```

A factor is formed by taking a number, symbol, or function reference, and optionally preceding it by a string of pluses and minuses. Each + is ignored, and each - alternates the resulting sign. A term is formed by combining factors with the operators ** (exponentiation), * (multiplication), and / (division). An expression is formed by combining terms with the operators + and -. Translation is done in a left to right scan, with operators being applied in the following order:

1. +, -   (unary)
2. **
3. *, /
4. +, -   (infix)

Thus the following expression generates intermediate results as indicated:

```
A + B*-C/D**E*F - X**Y**Z**-W/L/M*N
```

```
A

A, B

A, B, -C

A, B*(-C)

A, B*(-C), D

A, B*(-C), D, E

A, B*(-C), D**E

A, (B*(-C))/(D**E)

A, (B*(-C))/(D**E), F

A, ((B*(-C))/(D**E))*F

A+(((B*(-C))/(D**E))*F)

A+(((B*(-C))/(D**E))*F), X

A+(((B*(-C))/(D**E))*F), X, Y

A+(((B*(-C))/(D**E))*F), X**Y

A+(((B*(-C))/(D**E))*F), X**Y, Z

A+(((B*(-C))/(D**E))*F), (X**Y)**Z

A+(((B*(-C))/(D**E))*F), (X**Y)**Z, -W

A+(((B*(-C))/(D**E))*F), ((X**Y)**Z)**(-W)

A+(((B*(-C))/(D**E))*F), ((X**Y)**Z)**(-W), L

A+(((B*(-C))/(D**E))*F), (((X**Y)**Z)**(-W))/L

A+(((B*(-C))/(D**E))*F), (((X**Y)**Z)**(-W))/L, M

A+(((B*(-C))/(D**E))*F), ((((X**Y)**Z)**(-W))/L)/M

A+(((B*(-C))/(D**E))*F), ((((X**Y)**Z)**(-W))/L)/M, N

A+(((B*(-C))/(D**E))*F), (((((X**Y)**Z)**(-W))/L)/M)*N

(A+(((B*(-C))/(D**E))*F))-(((((X**Y)**Z)**(-W))/L)/M)*N)
```

If *quan* is a unit number, INLIST will read one line at a time from the current position, beginning with the next complete line, and ending with the line which terminates the input expression. If the expression is terminated by a $ within columns 7 through 72, the remaining columns may be used for a comment. If the expression is terminated by non-blank characters in columns 1 through 5 of a line following the expression, the entire line may be used for a comment, as long as at least one of columns 1 through 5 is not blank.

If *quan* is a list, INLIST assumes that each of the items on *quan* is a list of Hollerith constants, each such constant representing ten consecutive columns of a line. The list *quan* and each of its sublists is scanned from top to bottom, in effect "rewinding" *quan* with each call to INLIST. Thus, only the first expression on *quan* is translated, and everything beyond the terminator of this expression may be used for a comment.

The following sets of line images and calls to LSQMNL will be translated by INLIST into valid representations of the indicated expressions:

```
[[0]]          0$ THIS IS A CØMMENT                       (file)
               LSQMNL(LSQMNL(10H      0$  ))              (list)
[[x - y^z]]    X - Y**Z                                   (file)
           * THIS IS ALSØ A CØMMENT
               LSQMNL(LSQMNL(10H      X - ,10HY**Z      ), (list)
                          LSQMNL(1H*))
```

$[[3\sqrt{z} + 5x/y - (x + y)^{3r}]]$

```
3*Z**.5 +                                          (file)
*   5*X   /   Y
-(X - -Y)**+(3*R)
$END ØF EXPRESSIØN
LSQMNL(LSQMNL(1H ,7H3*Z**.5,1H+,3H5*X),  (list)
       LSQMNL(1H ,4H/Y-(),
       LSQMNL(1H ,10HX - -Y)**+,5H(3*R)))
```

There are other possible forms of input for INLIST. These are discussed in detail in §3.2. One, which insures compatibility with some of the forms of expression output provided by the routine LSQPNT (§2.3), will be mentioned here. If, instead of being terminated, an expression is followed by an =, the expression is discarded, and the scan for an expression begins anew. Thus, an input line of the form

```
LA(X.Y, Z)=0=5$
```

will be translated into a valid representation of [[5]]. Though the expressions before the last equals sign are effectively comments, they must conform to the syntax for input expressions.

Exercises

2.2.1. Which of the following are valid symbols for INLIST?

.A	FUNCTIØN	A+B/C	DIMENSIØN
A.	1.*	E5	DIMENSIØNED
A.1E+5	(A.........)	1.E5	A.........

2.2.2. Which of the following are valid numbers for INLIST?

.	1.0	1234567890.0987654321
.0	1.1E-5	E-5
.1	3.14159E--5	.1D3

2.2.3. Consider the following input stream to lie on a file. Assume INLIST to be called five times for that file. What lists would be formed? What if the file were a list?

```
          LA=IF(A.EQ.B)+SIN(5.E-1)/3**2$XYZ$$
          X.......+Y......(X....**3)**X.+X.......
     $
          HØW=NØW=BRØWN=CØW(BULL)$ MY
          +GØØD/NESS.ØR.A
          $
          FUNC(ARG1,ARG2,
              SIN(ARG3)**3*X,
                          ARG4)+(((((0)))))
     $
```

2.3. Output of Expressions

The functions LSQPNT and LSQØUT provide a means of translating expressions from their internal representation as lists to a FORTRAN like notation. LSQPNT performs the translation to the level of Hollerith constants, and LSQØUT formats these into line images on a file or a list.

LSQØUT(4HUNIT, *quan*) directs all further output to the logical unit *quan*. For example,

```
CALL LSQØUT(4HUNIT, 6LØUTPUT)
```

would direct lines to the file ØUTPUT. LSQØUT assumes this unit as a default.

LSQØUT(4HLIST, *lis*) directs all further output to the list *lis*. As each line is completed it is broken into Hollerith constants representing consecutive groups of ten columns. These constants are then placed on a list, with columns 1 through 10 at the top. This newly created list is placed on the bottom of *lis*.

LSQPNT(*lis*, *hol*) translates the expression *lis* into line images. The first line begins with

hol =

in column 7, and the last two lines are of the form

```
$
$END ØF EXPRESSIØN
```

with the $ in column 2. Beginning after the equals sign on the

first line, columns 7 through 72 of the line images are used for a FORTRAN like representation of *lis*, and possibly of some of its subexpressions, conforming to the syntax given for INLIST in §2.2.

LSQPNT(0, 5HNØSUB) causes LSQPNT to translate all subexpressions where they occur. If this call is made, then LSQPNT will translate an expression into a single FORTRAN like expression. Otherwise, some deeply embedded subexpressions may be replaced by symbols of the form N.0, N.1, N.2, etc., with these subexpressions translated after the main expression.

LSQPNT(0, 3HSUB) undoes the effect of LSQPNT(0, 5HNØSUB). This is the default state of LSQPNT. It makes output expressions more readable, but is in conflict with normal INLIST usage.

In most cases, expressions take less list space as line images than in their internal representation. The user may take advantage of this fact to compress infrequently used expressions as follows:

```
CALL LSQPNT(0,5HNØSUB)
        .
        .
        .
LSAVE=LIST(9)
CALL LSQØUT(4HLIST,LSAVE)
CALL LSQPNT(LXPR,4HLXPP)
CALL LSQDES(LXPR)
        .
        .
        .
LXPR=INLIST(LXPR,LSAVE)
```

The code first conditioned LSQPNT to translate subexpressions where they occur. This need be done only once. Then the list LSAVE was created to hold the line images of the expression LXPR. LSQPNT was called to dump LXPR to LSAVE. Then LXPR was destroyed. Later it was recreated by a call to INLIST.

If no such special use of LSQPNT has been made, an expression may be printed simply by a call such as

```
CALL LSQPNT(LXPR,6HMYLIST)
```

but if such use has been made one should first use

```
CALL LSQØUT(4HUNIT,6LØUTPUT)
```

to insure that output will go to the file ØUTPUT.

Exercise

2.3.1. Run the following program on the data in exercise 2.2.3. Add to the data such expressions as may interest you. Compare the output with the input. (Set a short time limit to make the program stop.)

```
      PRØGRAM TEST(INPUT,ØUTPUT)
      DIMENSION SPACE(5000)
      CALL INITAS(SPACE,5000)
1     CALL INLIST(LA,5LINPUT)
      CALL LSQPNT(LA,2HLA)
      GØ TØ 1
      END
```

2.4. Destruction of Lists and Expressions

The functions LSQDES and LSQUNM unbuild lists and expressions.

LSQDES(*lis*) erases the list or expression *lis*. Lists that are no longer needed must be erased so that their cells will be available for other lists. If *lis* is a sublist of another list, it will not actually be destroyed. However, the user should consider *lis* unavailable in all cases, since a sublist is erased when the list of which it is a sublist is destroyed.

LSQUNM(*lis*, var_1, var_2, ..., var_k), k = 1, 2, ..., 59, removes k items from the bottom of list *lis*, the bottom item going to var_k, the next to var_{k-1}, etc. For example, an expensive way of doing very little is:

```
LA=LSQMNL(X,Y,Z,W,L,M)
CALL LSQUNM(LA,X,Y,Z,W,L,M)
```

This leaves LA as an empty list. LSQUNM returns *lis* as its value. Thus a way of doing effectively nothing is:

```
CALL LSQDES(LSQUNM(LSQMNL(A,B),A,B))
```

which creates a list with A and B on it, pops them off and destroys the list.

Exercises

2.4.1. What does the following code do?

```
CALL LSQDES(LSQUNM(LSQMNL(A,B,C),B,C,A))
```

2.4.2. The program given in exercise 2.3.1 contains an error. Correct it. (Hint: What would happen if the data produced 3000 expressions?)

2.4.3. Which lists are available after the following code is executed?

```
LA=LIST(9)
LB=LIST(9)
LC=LIST(9)
LD=LIST(9)
CALL MANY(LA,LB,LC,LD,LD,LC,LB)
CALL LSQDES(LB)
CALL LSQDES(LD)
CALL LSQUNM(LA,LF,LD,LC,LB)
CALL LSQDES(LA)
```

2.5. Other Output Facilities

The function PRLSTS provides a mechanism for printing dumps of lists. This is primarily for debugging purposes, when lists have been created to represent expressions, but cannot be printed by LSQPNT due to some error.

PRLSTS(*lis*, *int*) prints the list *lis* in the mode determined by the integer quantity *int*, which may be 1, 2, 3, or 4, for integer, Hollerith, real, or octal dumps respectively. Each dump begins a new page. The most likely call is

```
CALL PRLSTS(LXPR,4)
```

which will dump LXPR in octal.

```
BEGIN LIST
BEGIN SUBLIST
  [illegible]
END SUBLIST
BEGIN SUBLIST
  [illegible]
  [illegible]
  [illegible]
END SUBLIST
 BEGIN SUBLIST
  [illegible]
  [illegible]
  [illegible]
END SUBLIST
BEGIN SUBLIST
  [illegible]
  [illegible]
BEGIN SUBLIST
BEGIN SUBLIST
  [illegible]
  [illegible]
  [illegible]
END SUBLIST
END SUBLIST
  [illegible]
  [illegible]
END SUBLIST
END LIST
```

Figure 2.6.1. Output from a call to PRLSTS(LIS,4) with LIS representing the expression $[[1 + x + x^2 +$ [illegible]$]]$

2.6. Arithmetic Routines

The functions LSQADD, LSQMEX, and LSQRAZ are available to perform arithmetic operations on pairs of expressions. Each returns an expression as its value, and does not change its arguments.

LSQADD(lis_1, lis_2) returns the sum of expressions lis_1 and lis_2

LSQMEX(lis_1, lis_2) returns the product of expressions lis_1 and lis_2.

LSQRAZ(lis_1, lis_2) returns the result of raising the expression lis_1 to the expression lis_2 power.

In order to perform subtraction or division, addition of the negative or multiplication by the inverse respectively must be used. For example:

```
LAMINS=LSQMNL(LSQMNL(-1.))
LCM=LSQMEX(LAMINS,LC)
LCI=LSQRAZ(LC,LAMINS)
LDD=LSQADD(LB,LCM)
LDQ=LSQMEX(LB,LCI)
```

leaves LB-LC in LDD and LB/LC in LDQ.

Neither LSQADD nor LSQMEX introduce parentheses. The results are obtained by combining the terms of the operands to form new terms. For example, in the following, LDD will be an empty expression, since all its terms will cancel.

```
LAMINS=LSQMNL(LSQMNL(-1.))
LCM=LSQMEX(LAMINS,LC)
LDD=LSQADD(LC,LCM)
```

LSQRAZ does introduce parentheses in many cases. A sum of terms raised to some non-trivial power will not be expanded. However a single term raised to a power will have that power distributed over its factors. Thus, in the following example, LDQ will be the list ((1.)), representing [[1]], if LC is a single term, but not necessarily in any other case.

```
LAMINS=LSQMNL(LSQMNL(-1.))
LCI=LSQRAZ(LC,LAMINS)
LDQ=LSQMEX(LC,LCI)
```

Exercises

2.6.1. Write code to generate the following arithmetic operations on expressions LA, LB, LC, LD, etc.

LA+LB	LA+LB*LC	LA*LB+LC*LD+LE*LF
LA-LB	LA-LB**LC	LA**2+LB**2+LC**2
LA*LB	3*LA	LA/(1-LB**2)**.5
LA/LB	LA**3	LA*LB*LC*LD*LF*LG*LH

2.6.2. Devine the purpose of the following function

```
      FUNCTIØN LSQSVC(LA,N)
      DIMENSIØN LA(N)
      LSQSVC=LA(1)
      DØ 100 J=2,N
      LB=LSQSVC
      LSQSVC=LSQADD(LSQSVC,LA(J))
      IF(J.GT.2)CALL LSQDES(LB)
  100 CØNTINUE
      RETURN
      END
```

2.6.3. Write a subroutine LSQMMT to multiply two 2 by 2 matrices. - two 3 by 3 matrices. - two N by N matrices.- an N by M matrix by an M by K matrix.

2.7. Definition and Evaluation

The functions LSQVAL and LSQDEF provide facilities for evaluating expressions in terms of user provided definitions of expression symbols. The term "evaluation" refers to the application of all currently known definitions to an expression. It is proper to leave symbols undefined. Thus this process always returns an expression, not a FORTRAN constant. The routine LSQTYP (see §2.10) may be used to obtain a FORTRAN constant from a constant expression.

As part of standard program initialization (see LSQINI and INITAS in §2.11) certain definitions are made. If these and subsequent user definitions are suspended, temporary definitions may be introduced so that evaluation will effect substitution. Such a mechanism is provided by the routine LSQSBS. The calls to LSQDEF and LSQVAL used by LSQSBS are among those described in §3.6, where the precise definition and evaluation methods used are presented.

LSQVAL(*lis*) returns the expression representing the value of expression *lis*, obtained by applying all currently known definitions to the expression symbols of *lis*. For example, unless CØS is re-defined, the following code will leave ((1.)) in LAV, representing [[1]].

```
LA=LSQMNL(LSQMNL(1.,LIST(9),3HCØS,1.))
LAV=LSQVAL(LA)
```

The input expression *lis* (e.g. LA above) is unchanged.

LSQDEF(*symb*, *lis*, 0) defines the symbol *symb* to be the expression *lis*. The expression *lis* should be considered erased by this call. Where the symbol *symb* is used as a function name, its arguments will be applied to the expression symbols of *lis*. For example

```
      LA=LSQMNL(LSQMNL(1.,LSQMNL(LSQMNL(1.,1HX,1.)),
     $  5HSIGMA,1.))
      CALL LSQDEF(5HSIGMA,LSQMNL(LSQMNL(1.),
     $  LSQMNL(1.,3HTAU,1.)),0)
      LAV=LSQVAL(LA)
```

will leave a representation of $[[1 + \tau(x)]]$ in LAV.

LSQDEF(*symb*, lis_1, lis_2) defines the symbol *symb* to be the expression lis_1, in which the expression symbols on the list lis_2 hold the places into which any arguments applied to *symb* are to be substituted. Both lis_1 and lis_2 should be considered to be erased by this call. The list of dummy arguments lis_2 should be a list of expression symbols not used in any other such call to LSQDEF. In the course of an expression evaluation in which *symb* is encountered, the arguments to which it is applied will be scanned from left to right, while lis_2 is scanned from top to bottom, temporarily defining each expression symbol found on lis_2 to be the corresponding argument. The value of lis_1 subject to these definitions is used in place of *symb* and those of its arguments which have been matched. Any unmatched actual arguments are applied to the result. Though it is not an error to leave some dummy arguments unmatched, the user is advised to read §3.6 before so doing.

As an example of this call consider

```
CALL LSQDEF(3HSIN,LSQMNL(LSQMNL(1.,
$  LSQMNL(LSQMNL(1.),LSQMNL(-1.,LSQMNL(
$  LSQMNL(1.,3HL..,1.)),3HCØS,2.)),3H1.*,.5)),
$  LSQMNL(3HL..))
```

which defines [[*sin*(*l*..)]] to be [[$\sqrt{(1 - cos(l..)^2}$]].

LSQSBS(lis_1, *symb*, lis_2, -1) returns the result of substituting lis_1, an expression, for *symb*, an expression symbol, in the expression lis_2. All appearances of *symb*, with or without arguments will be replaced. An infinite loop may occur if lis_1 contains *symb*. The expression lis_1 should be considered erased by this call.

Exercises

2.7.1. Write a subroutine which when called will define 3HCØT as the cotangent function, 3HSEC as the secant function, and 3HCSC as the cosecant function. Assume that 3HSIN is already defined as the sine function, 3HCØS as the cosine function, and 3HTAN as the tangent function. Call the subroutine LSQTRG.

2.7.2. Discover the purpose of the following program.

```
      PRØGRAM TEST(INPUT,ØUTPUT)
      DIMENSIØN SPACE(5000)
      CALL INITAS(SPACE,5000)
      LDETR=LSQMNL(LSQMNL(1.,3HL11,1.,3HL22,1.),
     *   LSQMNL(-1.,3HL12,1.,3HL21,1.))
      CALL LSQDEF(3HL11,INLIST(LA,5LINPUT),0)
      CALL LSQDEF(3HL12,INLIST(LA,5LINPUT),0)
      CALL LSQDEF(3HL21,INLIST(LA,5LINPUT),0)
      CALL LSQDEF(3HL22,INLIST(LA,5LINPUT),0)
      LVDETR=LSQVAL(LDETR)
      CALL LSQDES(LDETR)
      CALL LSQPNT(LVDETR,6HDETERM)
      CALL EXIT
      END
```

2.8. Truncation

The function LSQTRC provides a mechanism whereby expressions may be formed in which some of the powers of a particular variable have been removed. The usual application is to power series, say [[$c_0 + c_1x + c_2x^2 + \ldots + c_mx^m + \ldots$]], in which the variable x is sufficiently small to warrant the assumption that powers beyond, say, the kth may be treated as zero. Then one would use the polynomial [[$c_0 + c_1x + c_2x^2 + \ldots + c_kx^k$]] in place of the power series. In this section we restrict our attention to this simple case of polynomials in the variable on which trucation is to be performed. For the action of LSQTRC on more complex expressions see §3.7.

LSQTRC(*lis*, *symb*, *quan*) returns an expression in which those terms of *lis*, a polynomial in the expression symbol *symb*, which contain *symb* raised only to a constant power between zero and *quan*, inclusive, are retained. For example

```
       LA=LSQMNL(LSQMNL(.5),LSQMNL(3.,1HX,21.),
      $  LSQMNL(-6.,1HX,44.,1HY,1.))
       LA1= LSQTRC(LA,1HX,0.)
       LA2= LSQTRC(LA,1HY,0.)
       LA3= LSQTRC(LA,1HZ,0.)
       LA4= LSQTRC(LA,1HX,21.)
```

will leave representations of [[.5]], [[$.5 + 3x^{21}$]], [[$.5 + 3x^{21} - 6x^{44}y$]], and [[$.5 + 3x^{21}$]] in LA1, LA2, LA3, and LA4 respectively.

LSQTRC(*lis*, *symb*, $quan_1$, $quan_2$) returns an expression in which those terms of *lis*, a polynomial in the expression symbol *symb*, which either contain *symb* raised to a constant power between $quan_1$ and $quan_2$ inclusive, or, if zero lies in that range, do not contain *symb* at all, are retained. The order of $quan_1$ and $quan_2$ is not material, i.e., LSQTRC(LA,1HX,3.,5.) returns the same expression as LSQTRC(LA,1HX,5.,3.).

Exercises

2.8.1. The following function will return the coefficient of the expression symbol SYMB raised to the power QUAN in the polynomial LIS. Use this function to write a program which will differentiate polynomials of known degree.

```
FUNCTIØN LSQGCØ(SYMB,QUAN,LIS)
LA=LSQTRC(LIS,SYMB,QUAN,QUAN)
LB=LSQMNL(LSQMNL(1.))
LSQGCØ=LSQSBS(LB,SYMB,LA,-1)
CALL LSQDES(LA)
RETURN
END
```

2.8.2. Write functions LSQADT and LSQMUT which will add and multiply a pair of polynomials while truncating on some expression symbol to a specified power.

2.9. Differentiation

The functions LDIF and LOOKUP provide a mechanism for taking the derivatives of expressions with respect to expression symbols using user provided definitions of the partial derivatives of functions. By differentiation we mean the application of the following rules, for a particular expression symbol *symb*:

$$D_{symb}(a_1 + a_2 + \ldots + a_k) \rightarrow D_{symb}(a_1) + \ldots + D_{symb}(a_k)$$

$$D_{symb}(a_1 a_2 \ldots a_k) \rightarrow (D_{symb}(a_1)/a_1 + \ldots + D_{symb}(a_k)/a_k)a_1 a_2 \ldots a_k$$

$$D_{symb}(a^b) \rightarrow D_{symb}(a)ba^b/a + D_{symb}(b)a^b log(a)$$

$$D_{symb}(a(e_1,e_2,\ldots,e_k)) \rightarrow D_{symb}(e_1)partial(a(e_1,e_2,\ldots,e_k),1) + \ldots + D_{symb}(e_k)partial(a(e_1,e_2,\ldots,e_k),k)$$

$$D_{symb}(symb') \rightarrow \begin{cases} 1, \text{ if expression symbol } symb' \text{ is } symb \\ 0, \text{ if expression symbol } symb' \text{ is not } symb \end{cases}$$

$$D_{symb}(cons) \rightarrow 0, \text{ for any constant expression } cons$$

where the function *partial* is determined by user definitions via LOOKUP. It is proper to differentiate an expression involving functions for which some or all partial derivatives are not defined. In such a case, the expression symbol 7HPARTIAL will be used as the function name for *partial*.

LDIF(*lis*, *symb*) returns the derivative of expression *lis* with respect to expression symbol *symb*. The resulting expression

is the derivative in the sense of the above rules, in effect a partial derivative with respect to *symb*, since only explicit functional dependence is considered. For example, if LA represents

$$[[xy + log(-x) + x(y)]]$$

then LDIF(LA,1HX) will return a representation of

$$[[y - 1/x]]$$

since $x(y)$ does not depend on x.

LØØKUP(0, *symb*, *lis*, *int*) defines the expression *lis* to be the partial derivative of the function represented by the expression symbol *symb*, with respect to the argument *int*, an integer quantity. Arguments are numbered from the left, beginning with 1. Where an actual argument of *symb* is to be used in the partial derivative, the function ARG.(k) should be used in *lis*, with k as the number of the argument. The expression *lis* should be considered erased by this call. For example, the partial derivatives defined in standard initialization, see §2.11, include the following.

```
      CALL LØØKUP(0,3H1.*,LSQMNL(LSQMNL(1.)),1)
      CALL LØØKUP(0,3HSIN,LSQMNL(LSQMNL(1.,
     $  LSQMNL(LSQMNL(1.,LSQMNL(LSQMNL(1.)),4HARG.,1.)),
     $  3HCØS,1.)),1)
```

This defines the obvious derivatives of the identity function and the sine.

Exercises

2.9.1. Apply the following program to the indicated expressions as data. Differentiate those expressions with respect to X by hand and compare with the output. (Use a short time limit.)

```
      PRØGRAM TEST(INPUT,ØUTPUT)
      DIMENSIØN SPACE(5000)
      CALL INITAS(SPACE,5000)
    1 CALL INLIST(LA,5LINPUT)
      LB=LDIF(LA,1HX)
      CALL LSQDES(LA)
      CALL LSQPNT(LB,5HDERIV)
      CALL LSQDES(LB)
      GØ TØ 1
      END
```

data record:

```
1$
X$
X**2$
X**N$
1/X**N$
SIN(A*X)/CØS(A*X)$
F(X,X,X,X,X,X,X,X,X,X)$
F(X,Y,Z,W)$
LØG(N*X)-N*LØG(X)$
($$ BAD INPUT TO TERMINATE RUN
```

2.9.2. Write a subroutine LSQGRD which will calculate the gradient of an expression with respect to an array of expression symbols.

2.9.3. The following function returns the Nth derivative of expression LIS with respect to expression symbol SYMB. Use it to write a program which calculates the first ten terms of the Taylor series of an expression about the origin, and form the indefinite integral of those terms.

```
      FUNCTIØN LSQNDR(LIS,SYMB,N)
      IF(N.GT.0) GØ TØ 1
      LSQNDR=LSSCPY(LIS)
      RETURN
 1    LN=N
      LYS=LIS
 2    LSQNDR=LDIF(LYS,SYMB)
      IF(LN.NE.N)CALL LSQDES(LYS)
      IF(LN.EQ.1)RETURN
      LN=LN-1
      LYS=LSQNDR
      GØ TØ 2
      END
```

2.10. Analysis of Lists and Expressions

The functions SEQRDR, SEQLR, and SEQLL provide a means of scanning lists and expressions element by element. Lists, expressions, and their elements may be further analyzed by use of the functions LSQTYP, LISTMT, LSQCMP, LSTEQL, LSQCNM, and EQUAL.

SEQRDR(*lis*) returns a quantity to be used as a "sequence reader" for the list or expression *lis*. A sequence reader contains sufficient information to allow SEQLR to find the next element to the right on the list, and for SEQLL to find the next element to the left on the list. As initially provided by SEQRDR, a sequence reader is at the head of the list, pointing to the right to the top element of the list, and to the left to the bottom element of the list. The functions SEQLR and SEQLL "advance" readers, so that each call allows elements further into the list to be seen.

SEQLR(*var*, *intvar*) returns the next element to the right (downwards) on the list or expression for which the variable or array element *var* contains a sequence reader. The integer variable or array element *intvar* is set to -1, 0, or +1 if the element returned is a quantity other than a sublist, a sublist, or the head of the list respectively. In addition, *var* is advanced to the right, so that it now points one element further to the right.

SEQLL(*var*, *intvar*) returns the next element to the left (upwards) on the list or expression for which the variable or array element *var* contains a sequence reader. The integer variable or

array element *intvar* is set to -1, 0, or +1 if the element returned is a quantity other than a sublist, a sublist, or the head of the list respectively. In addition, *var* is advanced to the left, so that it now points one element further to the left.

SEQLR and SEQLL may be used in any combination. It is proper to advance more than once around a list. It should be borne in mind that the head of a list is treated as an element, which for most practical purposes is an undefined quantity. Consider, as an example, the following code:

```
      SA=SEQRDR(LA)
100   SAR=SEQLR(SA,NR)
      IF(NR.NE.0) GØ TØ 100
```

where LA is a list. If LA contains any sublists then this code will leave SAR set to the first sublist, but will loop forever if LA does not contain any sublists. On the other hand, the following code will terminate with a count of the sublists of list LA in NSLA, and a count of all other elements in MELA. Note that sublists of sublists are not counted, nor are other elements of sublists.

```
      SA=SEQRDR(LA)
      NSLA=0
      MELA=0
100   SAR=SEQLR(SA,NR)
      IF(NR)101,102,103
```

```
101  MELA=MELA+1
     GØ TØ 100
102  NSLA=NSLA+1
     GØ TØ 100
103   ...
```

Thus, if LA were an expression, NSLA would contain the number of terms in LA and MELA would be zero.

LSQTYP(*quan*, *var*) returns -1 if the quantity *quan* is not a list (hence not an expression), and zero or greater if the quantity *quan* is a list or expression. The variable or array element *var* is set to the quantity *quan* if *quan* is not a list, otherwise it is used to provide additional information on the nature of *quan* considered as an expression. If *quan* is to be considered a list, but not an expression necessarily, then *var* will not contain useful information, and only the sign of the result returned by LSQTYP is significant.

If *quan* is a constant expression, then LSQTYP is zero, and *var* is set to the FORTRAN constant equivalent to *quan* (as a floating point number). As indicated in §2.7, this is the method by which numeric values for use in FORTRAN arithmetic expressions may be obtained from the results of application of LSQVAL for evaluation. For example, a rather expensive way of calculating the square root of 2 is given in the following code:

```
LA=LSQMNL(LSQMNL(1.,1HX,.5))
CALL LSQDEF(1HX,LSQMNL(LSQMNL(2.)),0)
```

```
LB=LSQVAL(LA)
NLA=LSQTYP(LB,XLA)
```

This leaves XLA set to $2^{.5}$, and NLA set to zero.

If *quan* is an expression of just one term, but is not constant, then LSQTYP will return a value ranging from 10000 through 79999. If *quan* is an expression of more than one term, say n terms, n greater than 1, then LSQTYP returns 80000+n, and *var* is set to 0.

If *quan* is an expression of just one term, but more than one factor, say n factors (exclusive of the coefficient), n greater than 1, then LSQTYP returns either 50000+n or 70000+n: the former when the coefficient is 1., the latter otherwise.

If *quan* is an expression of just one term which consists of a single factor (possibly with a non-trivial coefficient), where the expression may be considered to represent $[[a\ b^c]]$ or $[[a\ b(e_1,e_2,\ldots,e_k)^c]]$ with a as a numeric coefficient, then (taking k to be zero when no arguments are present) LSQTYP returns 10000+k, 20000+k, 30000+k, or 40000+k; the first when a and c are both 1., the second when a is 1. and c is not, the third when c is 1. and a is not, and the last when neither a nor c is 1.

In any case where *quan* is an expression of just one term, *var* is set to the expression symbol used as the variable or function name in the leftmost factor.

For example, in the following code NLA will be 10000, and NAMLA will be 1HX, since LA represents $[[x]]$.

```
LA=LSQMNL(LSQMNL(1.,1HX,1.))
NLA=LSQTYP(LA,NAMLA)
```

LISTMT(*lis*) returns zero if the list *lis* is empty and thus is the expression representing [[0]]. Otherwise a non-zero value (actually -1) is returned.

LSQCMP(lis_1, lis_2, 3HEXP) returns zero if expressions lis_1 and lis_2 are identical, -1 if lis_1 is lexicographically less than lis_2, and +1 if lis_2 is lexicographically less than lis_1. This ordering is defined in detail in §1.4. There are other calls to LSQCMP which may be used on fragments of expresssions such as terms and factors. These are covered in §3.8. For the moment it should be noted that two expressions may be equivalent by application of the distributive laws, but not considered identical by LSQCMP, since parentheses are significant. The major use for this call to LSQCMP is to provide a means of sorting expressions to avoid redundant computations.

LSTEQL(lis_1, lis_2) returns zero if the lists (or expressions) lis_1 and lis_2 are identical, and a non-zero quantity otherwise (again actually ±1). Unlike LSQCMP, LSTEQL may be applied to arbitrary lists, not just expressions, but does not provide an ordering for sorts.

LSQCNM($symb_1$, $symb_2$) returns zero if the expression symbols $symb_1$ and $symb_2$ are identical, -1 if $symb_1$ is lexicographically less

than $symb_2$, +1 if $symb_2$ is lexicographically less than $symb_1$. For eaxample, LSQCNM(1HA,1HB), LSQCNM(1HA,2HAB), LSQCNM(2HAA,3HAAB), and LSQCNM(1HA,2HAA) are all -1, while LSQCNM(4HAABA,4HAAAZ) is +1. LSQCNM is used to determine ordering of expressions throughout the package, and may be replaced by a user routine of similar effect if a different ordering is desired.

EQUAL($quan_1$, $quan_2$) returns zero if quantities $quan_1$ and $quan_2$ are identical as bit patterns in a machine word, and a non-zero quantity otherwise. EQUAL may be used on any quantities, including expression symbols, but will not provide an ordering as does LSQCNM. It should be noted that if the arguments to EQUAL are lists or expressions, the value returned will be zero only if the arguments are precisely the same list. Identical, but distinct lists would return a non-zero value. The value returned by EQUAL should only be used directly in an arithmetic IF statement, or simple logical tests, not as a floating point quantity in arithmetic statements.

Exercises

2.10.1. The following function returns the Nth element from the top of list LIS. Write a companion function LSQNEB which will return the Nth element from the bottom of list LIS.

```
          FUNCTIØN LSQNET(LIS,N)
          SA=SEQRDR(LIS)
          DØ 100 J=1,N
      100 CALL STRDIR(SEQLR(SA,M),LSQNET)
          RETURN
          END
```

2.10.2. Suppose LA is an expression and MLA=LSQTYP(LA,XLA). Describe LA for the following values of MLA and XLA.

10000,3H1.*	20005,5HFUNCT	30000,1HX	40001,3HCØS
0,36.5	-1,0	80120,0	30001,3HTAN
10001,3HSIN	0,0	50017,3H1.*	20001,4HTANH

2.10.3. Revise the programs in excerises 2.3.1 and 2.9.1 to cease input and call EXIT on the reading of a zero expression. - on the reading of an expression representing [[*end*]].

2.10.4. Write a function LSQDGR which determines the degree of a polynomial in expression symbol SYMB by differentiating to an expression which evaluates to zero. - by searching for the highest power of SYMB in the polynomial.

2.11. Initialization

The user has one initialization responsibility to SYMBØLANG. He must declare an array to be the list of available space from which list elements are to be drawn by a call to the routine INITAS. The routines of the package are in all other respects self-initializing. (There is an exception for special print definitions noted in §3.3).

The two most important examples of this self-initialization arise in evaluation and differentiation, where, on the first use of LSQVAL and LDIF, standard defintions are created by calling the routines LSQINI and LSQIDR. The user may also call these routines to restore standard definitions after he has changed some.

INITAS(*array*, *int*) establishes the array *array* of dimension specified by the even integer quantity *int* as the available space list. This call must be made before any list processing is done, including calls to LØØKUP, LSQDEF, LSQINI, or LSQIDR. This call should be made only once, since, as part of self-initialization, some routines establish lists to be used in all subsequent calls. These lists would be destroyed by a second call to INITAS. Very little can be done with an available space list of dimension less than 3500, and 5000 is more practical. Thus a typical program should begin

```
PRØGRAM TYP(INPUT,ØUTPUT)
DIMENSIØN SPACE(5000)
CALL INITAS(SPACE,5000)
```

LSQINI is called by LSQVAL on its first use to define the expression symbols 3HSIN, 3HCØS, 3HEXP, 3HLØG, 3HTAN, 6HARCTAN, 4HTANH, 3H1.*, 2HV., 2HQ., and 3HIF.. If the user wishes to redefine these expression symbols, it is pointless to do so before performing at least one evaluation. At any time the user may call LSQINI to reestablish the following definitions:

A. Simple functions of one evaluated argument

symbol	*use*	*meaning*
3HSIN	SIN(X)	the sine of X radians
3HCØS	CØS(X)	the cosine of X radians
3HEXP	EXP(X)	the exponential of X, i.e. e^X
3HLØG	LØG(X)	the natural logarithm of X, i.e. $log_e(X)$
3HTAN	TAN(X)	the tangent of X radians
6HARCTAN	ARCTAN(X)	the arctangent of X given in radians
4HTANH	TANH(X)	the hyperbolic tangent of X radians

B. Special functions

symbol	*use*	*meaning*
3H1.*	(X)	the expression symbol 3H1.* is the internal representation of parentheses. When used with one argument, the value of that argument is returned.
	(X,Y,...)	When used with more than one argument, the evaluates of those arguments are themselves enclosed in parentheses to form the value.

2HV. V.(X) the expression symbol 2HV. causes its argument to be evaluated twice, and returns this double evaluate as its value.

2HQ. Q.(X) the expression symbol 2HQ. returns its argument unevaluated. Thus V.(Q.(X)) will return the value of X.

3HIF. IF.(X,Y,...) the expression symbol 3HIF. provides a conditional expression facility. It takes its arguments unevaluated and, scanning from left to right, returns the first even numbered argument following an odd numbered argument which evaluates to zero. The value returned is an unevaluated argument. If there is an even number of arguments and no odd numbered argument evaluates to zero, zero is returned. If there is an odd number of arguments, the last argument is not tested; rather it is returned unevaluated as the value of 3HIF. if no other odd numbered argument evaluates to zero.

LSQIDR is called by LDIF to define derivatives of the expression symbols 3H1.*, 3HSIN, 3HCØS, 3HTAN, 3HCØT, 3HSEC, 3HCSC, 6HARCSIN, 6HARCCØS, 6HARCTAN, 6HARCCØT, 6HARCSEC, 6HARCCSC, 4HSINH, 4HCØSH, 4HTANH, 4HCØTH, 4HSECH, 4HCSCH, 7HARCSINH, 7HARCCØSH, 7HARCTANH, 7HARCCØTH, 7HARCSECH, 7HARCCSCH, 3HEXP, 3HLØG. This call is made on the first use of LDIF. The user may call LSQIDR to reestablish these definitions at any time.

symbol	*use*	*meaning and derivative defined*	
3H1.*	(X)	parentheses	1.
3HSIN	SIN(X)	sine	CØS(X)
3HCØS	CØS(X)	cosine	-SIN(X)
3HTAN	TAN(X)	tangent	SEC(X)**2
3HCØT	CØT(X)	cotangent	-CSC(X)**2
3HSEC	SEC(X)	secant	TAN(X)*SEC(X)
3HCSC	CSC(X)	cosecant	-CØT(X)*CSC(X)
6HARCSIN	ARCSIN(X)	arcsine	1/(1-X**2)**.5
6HARCCØS	ARCCØS(X)	arccosine	-1/(1-X**2)**.5
6HARCTAN	ARCTAN(X)	arctangent	1/(1+X**2)
6HARCCØT	ARCCØT(X)	arccotangent	-1/(1+X**2)
6HARCSEC	ARCSEC(X)	arcsecant	X**-1 * (X**2-1)**-.5
6HARCCSC	ARCCSC(X)	arccosecant	-X**-1 * (X**2-1)**-.5
4HSINH	SINH(X)	hyperbolic sine	CØSH(X)
4HCØSH	CØSH(X)	hyperbolic cosine	SINH(X)
4HTANH	TANH(X)	hyperbolic tangent	SECH(X)**2
4HCØTH	CØTH(X)	hyperbolic cotangent	-CSCH(X)**2

4HSECH	SECH(X)	hyperbolic secant	-SECH(X)*TANH(X)
4HCSCH	CSCH(X)	hyperbolic cosecant	-CSCH(X)*CØTH(X)
7HARCSINH	ARCSINH(X)	hyperbolic arcsine	1/(X**2+1)**.5
7HARCCØSH	ARCCØSH(X)	hyperbolic arccosine	1/(X**2-1)**.5
7HARCTANH	ARCTANH(X)	hyperbolic arctangent	1/(1-X**2)
7HARCCØTH	ARCCØTH(X)	hyperbolic arccotangent	-1/(X**2-1)
7HARCSECH	ARCSECH(X)	hyperbolic arcsecant	-X**-1 * (1-X**2)**-.5
7HARCCSCH	ARCCSCH(H)	hyperbolic arccosecant	-X**-1 * (X**2+1)**-.5
3HEXP	EXP(X)	exponential	EXP(X)
3HLØG	LØG(X)	natural logarithm	1/X

It should be noted that a large number of the trigonometric and hyperbolic functions whose derivatives are defined in LSQIDR do not have their values defined in LSQINI.

Exercises

2.11.1. The definition of 3H1.* established by LSQINI is not the best choice for all purposes. When (X,Y,Z,...) is intended to represent a vector this definition is suitable. On the other hand, for evaluation purposes in general, the following is a better definition:

```
      CALL LSQDEF(3H1.*,LSQMNL(LSQMNL(1.,10H$.1.*$.$.$,1.)),
     *   LSQMNL(10H$.1.*$.$.$))
```

Run the following program with and without this definition. Describe the difference in the output.

```
      PRØGRAM TEST(INPUT,ØUTPUT)
      DIMENSIØN SPACE(5000)
      CALL INITAS(SPACE,5000)
      LA=LIST(9)
      CALL LSQDES(LSQVAL(LA))
      CALL LSQDES(LA)
C INSERT DEFINITIØN ØF 3H1.* HERE TØ REPLACE STANDARD DEF
      CALL LSQDEF(3HCØS,LSQMNL(LSQMNL(1.,
     *   LSQMNL(LSQMNL(1.),LSQMNL(-1.,3HSIN,2.)),3H1.*,.5)),0)
C I.E. CØS = (1-SIN**2)**.5
 1    CALL INLIST(LA,5LINPUT)
      IF(LISTMT(LA).EQ.0)CALL EXIT
      CALL LSQPNT(LA,6HSOURCE)
      LB=LSQVAL(LA)
      CALL LSQDES(LA)
      CALL LSQPNT(LB,4HVAL1)
      LC=LSQVAL(LB)
```

```
CALL LSQDES(LB)
CALL LSQPNT(LC,4HVAL2)
CALL LSQDES(LC)
GØ TØ 1
END
```

data record:

```
CØS**2 + SIN**2$
CØS(X)**2 + SIN(X)**2$
CØS(X)SIN(X)$
$    ZERØ EXPRESSIØN TØ END INPUT
```

2.11.2. The definition of 3HIF. as a conditional expression makes it possible for the user to easily create recursive definitions of expression symbols. For example, the following code would define the expression symbol 9HFACTØRIAL applied to the argument N....1 to be the factorial of N....1, i.e. N....1*(N....1 - 1)*(N....1 - 2)*... ...*2*1.

```
 LAM=LSQMNL(LSQMNL(10H           ,10HV.(IF.(
*    ,10HN....1,1,  ,10HN....1-1,1,10H,FACTØRIAL,
*    10H(N....1-1),10H*N....1))  ))
 CALL LSQDEF(9HFACTØRIAL,INLIST(LA,LAM),LSQMNL(6HN....1))
 CALL LSQDES(LAM)
```

Write a program using IF. which will evaluate binomial coefficients by the rule:

$$ {}_{N}C_{K} = {}_{N-1}C_{K} + {}_{N-1}C_{K-1} \quad \text{for } 0 < K < N $$

$$ {}_{N}C_{0} = {}_{N}C_{N} = 1 $$

2.12. Examples

We now present some worked examples of SYMBOLANG programming.

2.12.1. In some applications it may be a nuisance to have to form expressions representing constants. So we write a function LSQCØN(QUAN) which will return an expression equivalent to the constant QUAN. There is actually very little to do. If QUAN is zero an empty list is the appropriate expression. Otherwise ((QUAN)) is the correct form.

```
FUNCTIØN LSQCØN(QUAN)
LSQCØN=LIST(9)
IF(QUAN.EQ.0)RETURN
CALL MANY(LSQCØN,LSQMNL(QUAN))
RETURN
END
```

2.12.2. Parentheses are a great blessing when one wants to control the organization of an expression, but often hinder such tasks as comparison and evaluation. The following function returns a version of the expression LIS in which parentheses have to a large extent been removed. Indeed parentheses will remain only for expressions to positive or negative fractional powers and to the power -1. unless a non-constant power is encountered.

```
FUNCTIØN LSQXPN(LIS)
CØMMØN AVSL,X(100)
ASSIGN 100 TØ LØC
```

```
      LYS=LIS
      CALL STRDIR(VISIT(LØC),LSQXPN)
      RETURN
100   SA=SEQRDR(LYS)
      LTU=LIST(9)
101   SAT=SEQLR(SA,N)
      IF(N.GT.0)CALL TERM(LTU)
      SATR=SEQRDR(SAT)
      LVU=LSQMNL(LSQMNL(SEQLR(SATR,N)))
102   LWU=LSQMNL(1.)
      NAR=0
106   CALL STRDIR(SEQLR(SATR,N),LYS)
      IF(N)103,104,105
105   CALL LSQDES(LWU)
      LSU=LSQADD(LVU,LTU)
      CALL LSQDES(LVU)
      CALL LSQDES(LTU)
      LTU=LSU
      GØ TØ 101
104   CALL MANY(X(100),NAR+1,SA,SATR,LTU,LVU,LWU)
      SAG=VISIT(LØC)
      CALL LSQUNM(X(100),NAR,SA,SATR,LTU,LVU,LWU)
      CALL MANY(LWU,SAG)
      GØ TØ 106
103   IF(LSQCNM(LYS,3H1.*).EQ.0)GØ TØ 107
110   CALL MANY(LWU,LYS)
      CALL STRDIR(SEQLR(SATR,N),LYS)
```

```
      IF(N)108,109,105
109   CALL MANY(X(100),SA,SATR,LTU,LVU,LWU)
      CALL STRDIR(VISIT(LØC),LYS)
      CALL LSQUNM(X(100),SA,SATR,LTU,LVU,LWU)
      IF(LSQTYP(LYS,LAS).NE.0)GØ TØ 108
      CALL LSQDES(LYS)
      LYS=LAS
108   CALL MANY(LWU,LYS)
      LWU=LSQMNL(LWU)
125   LMU=LSQMEX(LWU,LVU)
      CALL LSQDES(LWU)
      CALL LSQDES(LVU)
      LVU=LMU
      GØ TØ 102
107   IF(NAR.NE.1)GØ TØ 110
      CALL STRDIR(SEQLR(SATR,N),LYS)
      IF(N)111,112,105
111   LYS=LSQMNL(LSQMNL(LYS))
      GØ TØ 141
121   CALL MANY(X(100),SA,SATR,LTU,LVU,LWU)
      CALL STRDIR(VISIT(LØC),LYS)
      CALL LSQUNM(X(100),SA,SATR,LTU,LVU,LWU)
141   CALL LSQUNM(LWU,LGU)
      CALL LSQDES(LWU)
      LWU=LSQRAZ(LGU,LYS)
      CALL LSQDES(LGU)
```

```
      CALL LSQDES(LYS)
      MWU=LSQTYP(LWU,XWU)
      IF((MWU.GE.50000).ØR.(MWU.EQ.0))GØ TØ 125
      IF(LSQCNM(XWU,3H1.*).NE.0)GØ TØ 125
      IF(MØD(MWU,10000).NE.1)GØ TØ 125
      SAW=SEQRDR(LWU)
      SAW=SEQRDR(SEQLR(SAW,N))
      YS=SEQLL(SAW,N)
      IF(N.GE.0)GØ TØ 125
      IF(YS)116,117,118
117   CALL LSQDES(LWU)
      GØ TØ 102
116   M=-YS
      ASSIGN 121 TØ LAG
      GØ TØ 130
118   M=YS
      ASSIGN 122 TØ LAG
130   K=M
      SN=SEQLL(SAW,N)
      CALL STRDIR(SEQLL(SAW,N),LARG)
      LCRG=LARG
      LBRG=LSQMNL(LSQMNL(SEQLL(SAW,N)))
      IF(K.EQ.0)GØ TØ LAG,(121,122)
131   L=K
      K=K/2
      L=L-2*K
      IF(L.EQ.0)GØ TØ 132
```

```
      LDRG=LBRG
      LBRG=LSQMEX(LDRG,LCRG)
      CALL LSQDES(LDRG)
132   IF(K.EQ.0)GØ TØ 133
      LDRG=LCRG
      LCRG=LSQMEX(LDRG,LDRG)
      IF(LDRG.NE.LARG)CALL LSQDES(LDRG)
      GØ TØ 131
133   IF(LCRG.NE.LARG)CALL LSQDES(LCRG)
      GØ TØ LAG,(121,122)
122   X=YS-M
      IF(X.NE.0)GØ TØ 124
      CALL LSQDES(LWU)
      LWU=LBRG
      GØ TØ 125
124   LRU=LSQMNL(LSQMNL(1.,LARG,3H1.*,X))
      CALL LSQDES(LWU)
      LWU=LSQMEX(LRU,LBRG)
      CALL LSQDES(LRU)
      CALL LSQDES(LBRG)
      GØ TØ 125
121   X=YS+M
      IF(X.NE.0)GØ TØ 126
      CALL LSQDES(LWU)
      LWU=LSQMNL(LSQMNL(1.,LBRG,3H1.*,-1.))
      GØ TØ 125
126   LBRG=LSQMNL(LSQMNL(1.,LBRG,3H1.*,-1.))
```

```
      GØ TØ 124
      END
```

This code is, perhaps unfortunately, typical of the sort of code needed to derive one expression from another. Basically, one sets up a recursive loop, first through the terms of an expression accumulating a sum and within that loop another loop through the factors of each terms accumulating a product. In this case, the sum was accumulated in the expression LTU, and the product in the expression LVU, while the sequence reader SA was used to scan for each term of the expression LYS being expanded, and the sequence reader SATR was used to find the factors of each term. When a function argument or non-constant power is encountered the readers and expressions being used to accumulate results are saved on the public list X(100) and a VISIT is paid to the same loop to expand parentheses there. The place where the particular task of this loop is introduced is in the test at statement 103. If we changed that statement to a CØNTINUE we would have a proper routine for producing a copy of an expression which is guaranteed to have correct ordering.

2.12.3. A task the user is more likely to wish to code himself is the evaluation of the determinant of a symbolic matrix. The following program does this task by a recursive expansion in terms of minors. Rather than use scratch arrays, a list of struck columns is used. As written, the program is limited to 5 by 5 or smaller matrices, but the subroutine LSQDET which does the work is not.

program:

```
      PROGRAM TEST(INPUT,OUTPUT)
      DIMENSION LSRAY(5000),LRAY(5,5)
      CALL INITAS(LSRAY,5000)
      READ 20,NDIM
20    FORMAT(I1)
      CALL GETIN(LRAY,NDIM)
      END
      FUNCTION LSQDET(LRAY,NDIM)
      COMMON AVSL,X(100)
      DIMENSION LRAY(NDIM,NDIM)
      ASSIGN 100 TO LOC
      ASSIGN 200 TO IFIND
      LMI=LSQMNL(LSQMNL(-1,))
      LSTRUC=LIST(9)
      IORG=1
      JORG=1
      NSIZ=NDIM
      CALL STRDIR(VISIT(LOC),LSQDET)
      CALL LSQDES(LSTRUC)
      CALL LSQDES(LMI)
      RETURN
100   IF(NSIZ,GT,2)GO TO 1000
      MORG=IORG+1
      CALL VISIT(IFIND)
      LTU=LSQMEX(LRAY(IORG,JORG),LRAY(MORG,JORG+1))
      LTV=LSQMEX(LRAY(IORG,JORG+1),LRAY(MORG,JORG))
      LTW=LSQMEX(LMI,LTV)
      CALL LSQDES(LTV)
      LTX=LSQADD(LTU,LTW)
      CALL LSQDES(LTU)
      CALL LSQDES(LTW)
      CALL TERM(LTX)
1000  JORG=JORG+1
      NSIZ=NSIZ-1
      MORG=IORG+1
      CALL VISIT(IFIND)
      CALL MANY(LSTRUC,IORG)
      IORG=MORG
      CALL STRDIR(VISIT(LOC),LTU)
      MORG=IORG
      CALL LSQUNM(LSTRUC,IORG)
      LTW=LSQMEX(LRAY(IORG,JORG-1),LTU)
      CALL LSQDES(LTU)
      KSIZ=NSIZ
      LSIGN=-1
2000  KSIZ=KSIZ-1
      CALL MANY(LSTRUC,MORG)
      CALL MANY(X(99),KSIZ,LSIGN,LTW)
      CALL STRDIR(VISIT(LOC),LTU)
      CALL LSQUNM(X(99),KSIZ,LSIGN,LTW)
      CALL LSQUNM(LSTRUC,MORG)
      IF(LSIGN,GT,0)GO TO 2500
      LTM=LSQMEX(LMI,LTU)
      CALL LSQDES(LTU)
      LTU=LTM
2500  LSIGN=-LSIGN
```

```
      LTM=LSQMEX(LRAY(MORG,JORG-1),LTU)
      CALL LSQDES(LTU)
      LTU=LSQADD(LTW,LTM)
      CALL LSQDES(LTW)
      CALL LSQDES(LTM)
      LTW=LTU
      IF(KSIZ,LE,0)GO TO 3000
      MORG=MORG+1
      CALL VISIT(IFIND)
      GO TO 2000
3000   JORG=JORG-1
      NSIZ=NSIZ+1
      CALL TERM(LTW)
200    SA=SEQRDR(LSTRUC)
      KFLG=0
201    CALL STRDIR(SEQLR(SA,N),NORG)
      IF(N,GT,0)CALL TERM(0,)
      IF(NORG-MORG)201,202,203
202    MORG=MORG+1
      IF(KFLG,EQ,0)GO TO 201
      GO TO 200
203    KFLG=1
      GO TO 201
      END
      SUBROUTINE GETIN(LRAY,NDIM)
      DIMENSION LRAY(NDIM,NDIM)
      DO 100 J=1,NDIM
      DO 100 K=1,NDIM
100    LRAY(J,K)=INLIST(LRAY(J,K),5LINPUT)
         LEX=LSQDET(LRAY,NDIM)
      CALL LSQPNT(LEX,4HDETR)
      DO 200 J=1,NDIM
      DO 200 K=1,NDIM
200    CALL LSQDES(LRAY(J,K))
      CALL LSQDES(LEX)
      RETURN
      END
```

data record:

```
4
A11$
A12$
A13$
A14$
A21$
A22$
A23$
A24$
A31$
A32$
A33$
A34$
A41$
A42$
A43$
A44$
```

The data provided is a dimension between 2 and 5 (in this case 4) followed by the elements of each row of the matrix.

output:

```
DETR = A11*A22*A33*A44   -   A11*A22*A34*A43   -   A11*A23
   *A32*A44   +   A11*A23*A34*A42   +   A11*A24*A32*A43   -
   A11*A24*A33*A42   -   A12*A21*A33*A44   +   A12*A21*A34
   *A43   +   A12*A23*A31*A44   -   A12*A23*A34*A41   -
   A12*A24*A31*A43   +   A12*A24*A33*A41   +   A13*A21*A32
   *A44   -   A13*A21*A34*A42   -   A13*A22*A31*A44   +
   A13*A22*A34*A41   +   A13*A24*A31*A42   -   A13*A24*A32
   *A41   -   A14*A21*A32*A43   +   A14*A21*A33*A42   +
   A14*A22*A31*A43   -   A14*A22*A33*A41   -   A14*A23*A31
   *A42   +   A14*A23*A32*A41
$
$END OF EXPRESSION
```

3. BELLS, WHISTLES, AND FRILLS

The information in this section should not be needed for most SYMBOLANG applications. Here may be found alternative routines and calls to routines mentioned previously which may provide some coding convenience, or extra efficiency, or access to internal hooks. Furthermore, some routines which are not recommended for new code are included to assist in reading existing programs. Niceties, fine distinctions, special cases, and funny tricks (if any) are largely confined to this section. The programmer wishing to write code which involves detailed structural analysis of expressions, should find this material of interest.

3.1. Additional List and Expression Creation Methods.

In addition to the methods described in §2.1, calls to the functions CPYTRM, INLSTL, INLSTR, LIST, LSQCPY, LSQCXP, NEWBØT, NEWTØP, NULSTL, NULSTR, NXTLFT, NXTRGT, PUTLST, SUBSBT, SUBST, and SUBSTP provide means of constructing lists and expressions from scratch or from existing lists. Some of these calls could also be considered as methods of destroying lists, and will be mentioned again in §3.4.

CPYTRM(lis_1, lis_2) removes the top element from list or expression lis_1 and places it on the bottom of list or expression lis_2. If lis_1 is empty, so there is no top element to remove, then no change is made in either argument, and CPYTRM returns 0., else CPYTRM returns 1. as its value. If both arguments are expressions,

the effect is to subtract the leftmost term from lis_1 and add it to lis_2. Such use is not recommended since correct ordering is not necessarily preserved, and the destructive change to lis_1 may affect expressions of which it is a subexpression. This routine should not be used in new code.

INLSTL(*lis*, *quan*) removes all the elements from list or expression *lis* and inserts them to the left of the list element specified by the quantity *quan* as a sequence reader (see §2.10). The list element before which the insertion is to be made must reside on a list distinct from *lis*, and should be the next element to which the sequence reader *quan* would make a right advance. INLSTL returns the emptied list *lis* as its value. No attempt is made to change the quantity *quan*. It is not advanced as a reader, and thus will still point to the element to the right of the insertion on the right and to the element to the left of the insertion on the left. As in §2.10, the head of a list is considered an element of the list in this context. For example, a list (1, 2, 3, 4, 5, 6) may be created by the folowing code.

```
LA=LSQMNL(1,4)
LB=LSQMNL(2,3)
LC=LSQMNL(5,6)
SA=SEQRDR(LA)
SAC=SEQLR(SA,N)
CALL LSQDES(INLSTL(LB,SA))
```

```
SAC=SEQLR(SA,N)
CALL LSQDES(INLSTL(LC,SA))
```

This makes LA the lists (1,4), (1,2,3,4), and (1,2,3,4,5,6) in turn, while SA points to 1 at creation, to 4 on the first advance, and to the list head on the second advance.

In view of the destructive manipulations performed by INLSTL, its use on expressions should be avoided.

INLSTR(*lis*, *quan*) removes all the elements from the list or expression *lis* and inserts them to the right of the list element specified by the quantity *quan* as a sequence reader. The list element after which the insertion is to be made must reside on a list distinct from *lis*, and should be the next element to which the sequence reader *quan* would make a right advance. INLSTR returns the emptied list *lis* as its value. With obvious modifications, the remarks for INLSTL apply to this routine. The differences may be seen by considering the following code, which creates a list (1, 4, 2, 5, 6, 3).

```
LA=LSQMNL(1,4)
LB=LSQMNL(2,3)
LC=LSQMNL(5,6)
SA=SEQRDR(LA)
SAC=SEQLR(SA,N)
CALL LSQDES(INLSTR(LB,SA))
SAC=SEQLR(SA,N)
CALL LSQDES(INLSTR(LC,SA))
```

This makes LA the lists (1,4), (1,4,2,3), and (1,4,2,5,6,3) in turn, while SA points to 1 at creation, to 4 on the first advance, and to 2 on the second advance.

As with INLSTL, the use of INLSTR on expressions should be avoided.

LIST(0) returns a newly created empty list, (), which is the correct representation of [[0]]. This result agrees with that of LIST(9), except that the list thus created acts as if it were already a sublist of some other list, and will not be destroyed by either a call to LSQDES nor by the erasure of lists of which it is subsequently made a sublist. The function IRALST described in §3.4 must be used to erase a list created in this manner. Such a list is fairly well protected against accidental erasure. For example, it may still be referenced after use as an argument in a call to LSQDEF (see §2.7) or LOOKUP (see §2.9), though subsequent modification of the list by the user could have disastrous consequences. If an expression has been created by other means, and has not been manipulated beyond the point of creation, this call to LIST might be combined with a use of INLSTL to create a protected version of the expression. For example, if LA is an expression just formed by INLIST, the following code will have that effect.

```
LB=LIST(0)
CALL LSQDES(INLSTL(LA,SEQRDR(LB)))
```

This leaves LB as a protected version of LA, but destroys LA. On the other hand, a much safer way of protecting expressions is

available, even if one does not wish to explicitly make them sublists of a known, well protected list. If LA is a list or expression

```
CALL SETIND(-1,-1,LCNTR(LA)+1,LNKR(LA)+1)
```

will make the list or expression look as if it were a sublist of one more list than it actually is. Subject to the previous comments about using such protection at all, this last method may be used on any list or expression at any time after its creation and prior to its destruction. If this method is used, IRALST is needed to erase it rather than LSQDES.

LSQCPY(*var*) returns a newly created list which contains essentially the same elements as are pointed to by the variable or array element *var* treated as a sequence reader of some list or expression. The elements copied onto the newly created list are those which are reached by right advances of the sequence reader *quan* until it reaches the head of the list it is scanning. The sequence reader *quan* is itself advanced in the process. For example, the following code will leave LA as the list (1, 2, 3) and LB as the list (2, 3).

```
LEX=LSQMNL(1,2,3)
SA=SEQRDR(LEX)
SAC=SEQLR(SA,N)
LB=LSQCPY(SA)
```

```
LA=LSQCPY(SA)
```

LA is a full copy of the list LEX since SA was advanced all the way to the head of LEX in making the partial copy for LB. This routine may be used to properly break off terms from expressions. For example, in the following code the first term of expression LEX is placed on expression LA, and the remaining terms are placed on expression LB. LEX is not changed.

```
SA=SEQRDR(LEX)
SAT=SEQLR(SA,N)
LA=LIST(9)
IF(N.EQ.0)CALL MANY(LA,LSSCPY(SAT))
LB=LSQCPY(SA)
```

For elements which are sublists, LSQCPY will use the routine LSQCXP (see below) to either provide the sublist itself or a copy via LSSCPY. Thus LSQCPY may not provide a completely fresh copy of a list or expression, and sublists of the copy it returns should be treated with the same consideration afforded the sublists of the original. Normally LSQCXP does return the sublist itself, which implies that LSQCPY will return a copy made only on the first level having sublists in common with the original list.

LSQCXP(*lis*) either returns the list or expression *lis*, or returns a full copy of *lis* via LSSCPY. The choice is determined by the state of the first variable (usually called LSQCSX) in the

CØMMØN block labeled LSQCSX. If this variable is zero then the list itself is returned, else LSSCPY(*lis*). This variable is initialized by a DATA statement to zero. Thus, unless the user resets this variable to a non-zero value, LSQCXP will return the list or expression itself. LSQCXP is used by the other routines of the package which generate new expressions from old, such as LSQADD and LSQMEX, to determine when common terms and subexpressions are permitted. Thus, if the user wishes to prevent the introduction of common subexpressions he should use code of the form

```
CØMMØN/LSQCSX/LSQCSX
      .
      .
      .
LSQCSX=1
      .
      .
      .
```

This routine is largely for internal use of the package.

NEWBØT(*quan*, *lis*) places the quantity *quan* on the bottom of the list or expression *lis*. The value returned may be used as a sequence reader whose next right advance would bring it to the element *quan* just added to the bottom of *lis*. Until such a right advance is made, no left advance should be made since the reader begins with only a pointer to the right (to *quan*) but with no valid pointer to the left. Aside from the value returned MANY(*lis*, *quan*) has the same effect.

NEWTØP(*quan*, *lis*) places the quantity *quan* on the top of the list or expression *lis*. The value returned may be used as a sequence reader whose next right advance would bring it to the element *quan* just added to the top of *lis*. The same strictures apply to this sequence reader as to the value of NEWBØT.

NULSTL(*quan*, *lis*) returns a newly created list formed by stealing the elements to the right of the head of list or expression *lis* and to the left of the next element to be reached by a right advance of the quantity *quan* treated as a sequence reader, as well as stealing that element itself, provided there is such an element. The sequence reader *quan* must refer to elements of the list *lis*. If the element *quan* would reach by a right advance would be the head of the list *lis*, then no change is made in *lis* and an empty list is returned. Otherwise this constitutes a destructive change in *lis* and should normally be avoided for expressions. As an example of the effect of NULSTL consider the following code which leaves LA as the list (3, 4) and LB as the list (1, 2).

```
LA=LSQMNL(1, 2, 3, 4)
SA=SEQRDR(LA)
SAC=SEQLR(SA,N)
LB=NULSTL(SA,LA)
```

NULSTR(*quan*, *lis*) returns a newly created list formed by stealing the elements to the left of the head of the list or expression *lis* and to the right of the next element to be reached by a right advance of the quantity *quan* treated as a sequence reader, as well as stealing that element itself, provided there is such an element. The sequence reader *quan* must refer to elements of the list *lis*. If the element *quan* would reach by a right advance would be the head of the list *lis*, then no change is made in *lis* and an empty list is returned. Otherwise this is another destructive change in *lis* and should normally be avoided for expressions. As an example of the effect of NULSTR consider the following code which leaves LA as the list (1) and LB as the list (2, 3, 4).

```
LA=LSQMNL(1, 2, 3, 4)
SA=SEQRDR(LA)
SAC=SEQLR(SA,N)
LB=NULSTR(SA,LA)
```

NXTLFT($quan_1$, $quan_2$) inserts the quantity $quan_1$ as a new element in a list to the left of the element which would be reached by a right advance of the quantity $quan_2$ treated as a sequence reader.

NXTRGT($quan_1$, $quan_2$) inserts the quantity $quan_1$ as a new element in a list to the right of the element which would be reached by a right advance of the quantity $quan_2$ treated as a sequence reader.

PUTLST(lis, $quan^1{}_1$, $quan^1{}_2$, ..., $quan^1{}_{k_1}$, lis, $quan^2{}_1$, ..., $quan^2{}_{k_2}$, lis, ..., lis, $quan^i{}_1$, ..., $quan^i{}_{k_i}$, lis, ..., lis, $quan^r{}_1$, ..., $quan^r{}_{k_r}$, lis, lis) for $r = 1, 2, \ldots, k_1, k_2, \ldots, k_r = 1, 2, \ldots$; and up to 27 arguments in all; returns the expression lis, onto which the terms ($quan^i{}_1$, ..., $quan^i{}_{k_i}$) have been added. If these are valid terms the result is a valid expression. Thus each $quan^i{}_1$ must be a non-zero floating point constant, and so forth. For example, a valid representation of $[[x - y^3]]$ may be left in LA by the following code.

```
LA=LIST(9)
CALL PUTLST(LA,1.,1HX,1.,LA,-1.,1HY,3.,LA,LA)
```

The use of PUTLST should be avoided in new code.

PUTLST(0, lis, $quan^1{}_1$, ...) with arguments as above except for the leading 0, acts as the above call to PUTLST except that re-ordering of factors within the terms is prevented. This call should also be avoided in new code.

SUBSBT($quan$, lis) substitutes the quantity $quan$ for the bottom (rightmost) element of the list lis. It is an error to use this routine if the list lis is empty. The value returned

is the element which *quan* replaced. Since this is a destructive manipulation of *lis*, *lis* should not be an expression. The following code will leave LA as the list (1, 2, 4).

```
LA=LSQMNL(1, 2, 3)
CALL SUBSBT(4,LA)
```

SUBST($quan_1$, $quan_2$) substitutes the quantity $quan_1$ for the list element which would be reached by a right advance of the quantity $quan_2$ treated as a sequence reader. It is an error to use this routine if the element to be replaced is the head of a list. The value returned is the element which $quan_1$ replaced. The following code will leave LA as the list (1, 4, 3).

```
LA=LSQMNL(1, 2, 3)
SA=SEQRDR(LA)
SAC=SEQLR(SA,N)
CALL SUBST(4,SA)
```

SUBSTP(*quan*, *lis*) substitutes the quantity *quan* for the top (leftmost) element of the list *lis*. The same strictures apply to SUBSTP as to SUBSBT, and the value returned is again the element which *quan* replaced. The following code will leave LA as the list (4, 2, 3).

```
LA=LSQMNL(1, 2, 3)
CALL SUBSTP(4,LA)
```

Exercises

3.1.1. Devine the purpose of the following function. (Hint - LIS is an expression).

```
      FUNCTIØN LSQREM(LIS)
      SA=SEQRDR(LIS)
      SAT=SEQLR(SA,N)
      LSQREM=LSQCPY(SA)
      RETURN
      END
```

3.1.2. Run the following program. Discuss its sins.

```
      PRØGRAM SNAFU(INPUT,ØUTPUT)
      DIMENSIØN SPACE(5000)
      CALL INITAS(SPACE,5000)
      LA=INLIST(LA,5LINPUT)
      LB=INLIST(LB,5LINPUT)
      LC*LSQADD(LA,LB)
      LD=LSQMEX(LA,LB)
 1    CALL LSQPNT(LA,2HLA)
      CALL LSQPNT(LB,2HLB)
      CALL LSQPNT(LC,2HLC)
      CALL LSQPNT(LD,2HLD)
      SA=SEQRDR(LA)
      SAT=SEQLR(SA,N)
      SA=SEQRDR(SAT)
      CALL NXTRGT(LB,SA)
      CALL CPYTRM(LA,LB)
      GØ TØ 1
```

```
END
```

data record:

```
SIN(X)+CØS(X)-SIN(X)*CØS(X)**-1$
SIN(X)-CØS(X)+SIN(X)**-1*CØS(X)$
```

3.2. Additional Expression Input Capabilities

The function INLIST described in §2.2 is somewhat more liberal in accepting FORTRAN-like expressions for translation than we have indicated thus far. As was mentioned at the end of §2.2, the use of the "equals" sign, =, is permitted. This is achieved by using a slightly different syntax than was shown. The definition of an expression in the input stream is actually

<expression> → <expression part> { = <expression part>}$_0^\infty$

<expression part> → <term> {<addition operator><term>}$_0^\infty$

rather than

<expression> → <term> {<addition operator><term>}$_0^\infty$

which effectively adds = as a fifth class of operator to be applied after the infix addition operators + and -. This syntax is applied at all levels of the scan. Thus the equals sign may be properly embedded deep within an expression. For example, the following is valid input.

```
LA=1+SIN(ARG(A=B,C=D)=0)$
```

If the call to INLIST, INLIST(*var*, *quan*) is used, the expression parts to the left of an equals sign are discarded in translation to internal representation. In this case, the input above would be equivalent to

```
1+SIN(0)$
```

which makes the discarded expression parts little more than comments. There is another call to INLIST however, which allows the user to make use of the expression parts to the left of equals signs.

INLIST(*var*, *quan*, 3HVAL) returns a list representing the next group of expressions found on the logical unit or list *quan*. The variable or array element *var* will also contain the newly created expression. The scan of the input stream and the syntax applied are the same as for the two argument call to INLIST. However, groups of expressions are translated, rather than a single expression. The result returned is the value of the first expression in the group, with evaluation performed on this first expression after each of the following expressions, if any, has been evaluated and erased. The expressions considered are all those encountered in a scan of the input stream up to a double termination, i.e. two dollar signs appearing consecutively within columns 7 through 72 of lines of the input stream (with or without intervening blanks), or two consecutive lines which have columns 1 through 5 non-blank (and columns 7 through 72 blank), or some mixture of terminations. The first expression in a group begins a new line, but the remaining expressions in a group begin directly after the terminators of the prior expressions. For this purpose, termination of an expression by non-blank columns 1-5 counts as a dollar sign in column 7 of the terminating line, and a new

expression in the same group may begin with column 8 of the terminating line.

The translation of the equals sign differs from the two argument call to INLIST. In this three argument call to INLIST, expression parts to the left of equals sign are not ignored. Instead, input of the form

$$\ldots\ e_1 = e_2 = \ldots = e_k\ \ldots$$

is translated into

$$\ldots\ \mathrm{EQUAL.}(e_1,\ e_2,\ \ldots,\ e_k)\ \ldots$$

using the expression symbol EQUAL. (i.e. 6HEQUAL.), which is not normally defined by SYMBOLANG initialization (see §2.11). Thus the input

```
LA=1+SIN(ARG(A=B,C=D)=0)$$
```

is translated into the equivalent of the input

```
EQUAL.(LA,1+SIN(EQUAL.(ARG(EQUAL.(A,B),
*     EQUAL.(C,D)),0))$$
```

Note that two dollar signs are needed to terminate the scan.

By defining the expression symbol 6HEQUAL. via LSQDEF, the

user may cause this three argument call to INLIST to return any reasonable translation for the equals sign. One approach is to make EQUAL. a means of introducing definitions, defining a name on the left hand side to be the expression on the right hand side. Then the appearances of those names defined in the second and later expressions of a group would be replaced by their definitions in the first expression of the group. This technique is detailed in §3.6.

There is a function, LISTØN, which consists of nothing more than a two argument call to INLIST for input from the standard input unit (5LINPUT), i.e.

LISTØN=INLIST(*lis*,5LINPUT)

Thus, in order to read expressions from a data record in his input stream, the user may have code such as

LA=LISTØN(LA)

which will read an expression from current position to a single terminator.

3.3. Additional Expression Output Capabilities

The functions LSQPNT and LSQØUT described in §2.3 have additional calls available to facilitate meeting special output needs.

LSQPNT(0, 5HFLØAT) causes all subsequent output of non-zero numbers by LSQPNT to include a decimal point, even if indistinguishable from integers. Unless this call is made, numbers within expressions which have no fraction are printed without a decimal point. For example the list ((1.),(2.5,1HX,1.)) which represents [[1 + 2.5x]] would usually be printed as

```
        hol = 1  +  2.5*X
    $
    $END ØF EXPRESSIØN
```

but after this call would be printed as

```
        hol = 1. + 2.5*X
    $
    $END ØF EXPRESSIØN
```

which is useful in applications where SYMBOLANG is being used to prepare FORTRAN statements.

LSQPNT(0, 3HFIX) reverses the effect of a call to LSQPNT(0, 5HFLØAT) and allows decimal points to be dropped where

the number being output is an integer, i.e. has no fraction. Whether or not this call is made, the number 0 always prints without a decimal point.

LSQPNT(0, 5HNØPRE) causes all subsequent output of expressions to lack the leading

hol =

and the trailing "$" and "$END ØF EXPRESSIØN". Output will not necessarily begin on a new line. Rather it will follow on the same line as any other output via LSQØUT which has not been forced out of the internal buffer in LSQØUT either by filling the buffer or by a call to LSQØUT(5HFLUSH). The last line of the expression will not be forced out of the buffer in LSQØUT. This form of output is of very little value unless a call to LSQPNT(0, 5HNØSUB) has been made.

LSQPNT(0, 3HPRE) reverses the effect of a call to LSQPNT(0, 5HNØPRE), returning LSQPNT to its initial state in this respect.

LSQØUT(5HFLUSH) flushes the internal buffer in LSQØUT. If the buffer happens to be empty, this call has no effect. Thus, when LSQPNT has been used after a call to LSQPNT(0, 5HNØPRE) it should be followed by this call to LSQØUT.

LSQØUT(0, *hol*) outputs the characters to the left of the first blank in the Hollerith quantity *hol*. Up to ten characters may be output. If the leftmost character is a blank, this call to LSQØUT has no effect.

LSQØUT(*int*, *holarray*) outputs the leftmost *int* characters of the Hollerith string *holarray*, inclusive of blanks. The integer quantity *int* must be greater than zero.

LSQØUT(6HMARGIN, *int*) sets the left margin used by LSQØUT to the integer quantity *int*, which may range from 1 through the current value of the right margin. The left margin starts set to column 7, but is adjusted frequently by LSQPNT to indent lines of expression output.

LSQØUT(4HEDGE, *int*) sets the right margin used by LSQØUT to the integer quantity *int*, which may range from the current value of the left margin through 150. The right margin starts set to column 72, but is adjusted frequently by LSQPNT. LSQØUT formats lines in its buffer starting at the left margin, and may use columns up to and including the right margin, but in order to avoid splitting Hollerith quantities unnecessarily, rarely actually reaches the right margin.

LSQØUT(5HCFLAG, *int*, *hol*) causes column *int* to be used to mark continuation lines with the leftmost non-blank character

of the Hollerith quantity *hol*. The integer quantity *int* may range from 1 through 150, and need not lie within the margins. Whenever LSQØUT starts a new line because the current line is full (rather than because of a call to LSQØUT(5HFLUSH)) the continuation character is placed in column *int* of the new line. LSQPNT makes no attempt to override such a call, so continuation marks may be used in outputting expressions.

LSQØUT(5HCFLAG, 0) cancels the selection of a continuation mark. This is the normal state of LSQØUT.

LSQØUT($quan_1$, $quan_2$, ..., $quan_k$), k = 1, 2, ..., 60, has the same effect as sequential calls to LSQØUT. The argument list is scanned from left to right, grouping arguments to match the calls previously described. Thus one might set up for the punching of FORTRAN statements by the call

```
CALL LSQØUT(4HUNIT, 5LPUNCH, 6HMARGIN, 7,
*    4HEDGE, 72, 5HCFLAG, 6, 1H*)
```

These calls to LSQØUT and stacks of code within LSQPNT may be used to compose special output formats for particular expression symbols used as function names. For this purpose, the user must be aware of the following CØMMØN block definitions within LSQPNT

```
CØMMØN/LSQNTR/NTRYPT(10)
CØMMØN/LSQARG/LARG(10)
CØMMØN/LSQINT/INI
```

Assuming these definitions have been made, then the array NTRYPT will contain locations which the user may VISIT to output expressions, terms, or constants. The array LARG will be used to hold the lists or list fragments being processed. The variable INI is a trap location used in defining which expression symbols are to be handled by user code.

LSQPNT(INI, *symb*, *loc*) defines the location *loc*, established in an ASSIGN statement, to be the place for LSQPNT to visit when outputting the expression symbol *symb* as a function name. When the visit is made, LARG(6) will contain a sequence reader which points to the right to the first argument of *symb*.

The user may use the code to print an expression within LSQPNT while within his own code by storing the expression in LARG(1) and performing a VISIT to either NTRYPT(2) or NTRYPT(3), the former to print the expression without consideration as to whether it is a common subexpression or not, and the latter to take this factor into consideration. The code to output a constant may be used by storing the floating point constant in LARG(4) (without mode conversion) and performing a VISIT to NTRYPT(6).

The user returns control to LSQPNT by a call to TERM(0). Thus for example, the following code would cause the expression symbol 2HØR to print as an infix operator .ØR. (Let the user beware of confusion between the expression symbol 6HA.ØR.B and the result of printing ØR(A,B).)

```
          .
          .
          .
      CØMMØN AVSL,X(100)
      CØMMØN/LSQNTR/NTRYPT(10)
      CØMMØN/LSQARG/LARG(10)
      CØMMØN/LSQINT/INI
          .
          .
          .
      ASSIGN 100 TØ LAC
      CALL LSQPNT(INI,2HØR,LAC)
          .
          .
          .
100   CALL STRDIR(SEQLR(LARG(6),N),LARG(1))
101   CALL MANY(X(1),LARG(6))
      CALL VISIT(NTRYPT(3))
      CALL LSQUNM(X(1),LARG(6))
      CALL STRDIR(SEQLR(LARG(6),N),LARG(1))
      IF(N.NE.0)CALL TERM(0)
      CALL LSQØUT(4,4H.ØR.)
      GØ TØ 101
```

It should be noted that the sequence reader in LARG(6) had to be saved before using NTRYPT(3).

LSQPNT(INI, *symb*, 0) revokes any prior selection of special code to output the expression symbol *symb* as a function name.

LSQPNT(NTRYPT(1)) initializes LSQPNT sufficiently so that a user may store an expression in LARG(1) and a Hollerith quantity in LARG(2) and then VISIT NTRYPT(1) to completely simulate a call to LSQPNT(LARG(1),LARG(2)) even if the code is being executed within such a call. This allows definitions for LSQVAL to print expressions, yet for special printing definitions to use evaluation. (See §3.6).

There is a function, PRIPUT, which is used in some existing code to output expressions.

PRIPUT(*hol*, *lis*) is equivalent to a call to LSQPNT(*lis*, *hol*). This routine should not be used in new code.

Exercise

3.3.1. Write a program which will punch out the gradient of an expression as valid FORTRAN statements with sequential statement numbers, * in column 6 for continuation, and the original system of coordinates changed to X(1), X(2), etc.

3.4. Niceties of List and Expression Destruction

In addition to the methods described in §2.4, and the destructive effects of some of the routines in §3.1, the functions DELETE, IRALST, MTLIST, PØPBØT, PØPMID, PØPTØP, and RCELL unbuild lists and expressions. Normally, none of these functions, as well as LSQUNM, CPYTRM, INLSTL, INLSTR, NULSTL, and NULSTR, should be used to unbuild an expression. Indeed, other than erasure by a call to LSQDES, no change should ever be made to an existing expression without great care being taken, i.e. almost never. The reason for such caution is that most of the expression manipulation routines, including LSQVAL, LSQADD, and LSQMEX, freely incorporate sublists of the expressions they accept as input into the expressions they create as output. Thus any change to a sublist of one expression could well cause an undesired change in another expression. An even more extreme case is presented by expressions used in calls to LSQDEF and LØØKUP which may themselves appear in their entirety within created expressions. An expression which the user himself creates by use of LSQMNL, LIST, and INLIST (without the evaluative three argument call) may of course be manipulated freely prior to their use as arguments of true expression manipulation routines. Otherwise, if the user wishes to alter an expression, he must content himself to work with a fresh copy created by LSSCPY (<u>not</u> LSQCPY) in its place.

A user with some sort of death wish, who absolutely must take apart originals and not copies, may prevent the use of sublists of existing expressions in created expressions by using the CØMMØN

block LSQCSX to switch the function LSQCXP from returning its argument to returning LSSCPY of its argument. (See §3.1).

DELETE(*quan*) returns the list element which would be reached by a right advance of the quantity *quan* treated as a sequence reader, and removes that element from the list. It is an error to attempt to apply DELETE to the head of a list.

IRALST(*lis*) reduces the count of the number of times the list or expression *lis* appears as a sublist, and returns this reduced count as its value. If the reduced count is zero, *lis* is also erased. Unless the list *lis* was created by a call to LIST(0) this is the wrong way to erase it. Calling IRALST twice for the same list is a disaster.

MTLIST(*lis*) removes all the elements from the list or expression *lis*, and returns the now empty list *lis* consisting of just a list head as its value.

PØPBØT(*lis*) returns the bottom (rightmost) element of the list or expression *lis* as its value, and removes that element from the list. It is an error to apply PØPBØT to an empty list.

PØPMID(*var*) returns the list element on top of which the variable or array element *var* is "sitting" treated as a

sequence reader, removes that element from the list, and advances *var* to the left. Thus the next right advance of *var* will bring it to the same element as it was pointing to on the right originally. It is an error to apply PØPMID in such a way as to attempt to remove the head of a list. For example, *var* should never be a newly created sequence reader.

PØPTØP(*lis*) returns the top (leftmost) element of the list or expression *lis* as its value, and removes that element from the list. It is an error to apply PØPTØP to an empty list.

RCELL(*lis*) where *lis* is an empty list causes the absolute and total erasure of the list, no matter how many lists contain it as a sublist. Thus

```
CALL MTLIST(lis)
CALL RCELL(lis)
```

will erase *lis* despite any degree of protection. This is for emergencies only.

3.5. Further Arithmetic Operations on Expressions

The functions ADD, DVSUM, NUMPY, SUB, and SUMPY were used in some existing code to perform arithmetic operations on expressions. Their use in new code should be strenuously avoided.

ADD(lis_1, lis_2) returns the result of adding expression lis_1 to expression lis_2. The first expression is destroyed and the second is replaced by the sum.

DVSUM(lis_1, *symb*, lis_2) adds the result of dividing expression lis_1 by expression symbol *symb* to expression lis_2, replacing lis_2 with this result which is returned as the value of DVSUM.

NUMPY(*lis*, *quan*) replaces the expression *lis* with the result of multiplying this expression by the floating point constant *quan*, and returns the product as its value.

SUB(lis_1, lis_2) returns the result of subtracting expression lis_1 from expression lis_2. The first expression is destroyed and the second is replaced by the difference.

SUMPY(lis_1, lis_2, lis_3, $symb_1$, $quan_1$, ..., $symb_k$, $quan_k$), k = 0, 1, 2, ..., 5, returns the result of adding the product of expressions lis_1 and lis_2 to expression lis_3 and truncating on the expression symbols $symb_i$ to the powers $quan_i$.

3.6. Definition and Evaluation Revisited

The description of the functions LSQDEF and LSQVAL presented in §2.7 was limited to the simplest forms of definition and evaluation of expressions. The actual mechanism is rather complex. The functions which are available for definition and evaluation are LSQDEF, LSQDSF, LSQGAR, LSQSBS, LSQVAL, and LSQVVL. Some existing code contains references to the following older routines for evaluation, which should not be used in new code: EVALUE, HITENT, INSBST, INSUBT, INTENT, SBST, and SUBT.

One of the major concepts used in evaluation by LSQVAL is that of "level". Arguments of functions, expressions used as exponents, and definitions of expression symbols are considered to lie on a level one deeper than that of the expression on which they were discovered. The surface level, the level at which evaluation begins, is level 1; the arguments of functions in the expression being evaluated lie on level 2; arguments of functions within those arguments, on level 3; the definition of an expression symbol within these last arguments, on level 4; etc. Evaluation is performed recursively, in a left to right scan, always evaluating on level $n + 1$ before evaluating on level n. Naturally, when a constant expression is encountered, it is returned as its own value. When an undefined expression symbol without arguments is encountered, it too is returned as its own value. However, when an undefined expression symbol with arguments is encountered, its arguments are evaluated and the expression symbol applied to the evaluates. The user may set limits on the depth to which evaluation will be carried. Beyond

such a limit, evaluation will consist of simple copying.

LSQVAL(*lis*, *int*) returns the result of evaluating expression *lis* through level *int*, an integer quantity. If *int* is negative, no limit is in effect. If *int* is zero, a copy of the expression *lis* is returned. This copy may have common subexpressions with *lis*. The one argument call to LSQVAL(*lis*) is equivalent to LSQVAL(*lis*, -1), i.e. evaluation on all levels as in §2.7.

If simple substitutions of expressions for expression symbols, without any consideration of arguments, are the only definitions in effect, then limits on the level of evaluation will not result in great surprises. However, if dummy arguments get involved they may well appear in the evaluate if the limit level invokes the definition, since the dummy arguments could not be evaluated when encountered beyond the limit.

Since the concept of level does work well in simple substitutions, the function LSQSBS provides for its application.

LSQSBS(lis_1, *symb*, lis_2, *int*) returns the result of substituting expression lis_1 for the expression symbol *symb* in the expression lis_2 through the level specified by the integer quantity *int*.

In expressions where it is a priori known that beyond a certain level

the expression symbol for which a substitution is to be made does not appear, the user can greatly reduce the cost of substitution by using that level as a limit. If the limit level is 1, the expression symbol for which substitution is being made may appear safely within the expression to be substituted for the expression symbol without causing an infinite loop, since the evaluation will reduce to a copy as soon as the substitution is made.

The definitions used by LSQVAL also involve a concept of level. The same expression symbol may be defined on many different definition levels simultaneously, but only once on each level. Where we speak of the depth of a level within an expression, we shall speak of the height of a definition level. The lowest definition level is zero. This is the level on which the calls to LSQDEF described in §2.7 introduce definitions. There are further calls to LSQDEF which allow definitions on level zero in terms of FORTRAN code to VISIT, and definitions on higher levels. At any given time, the user may make a definition on level zero, introduce a new highest level of definition, make a simple substitution definition on the highest level, discard all the definitions on the highest level (provided that level is not zero) and regress to the next definition level down, suspend the use of level zero definitions and the right to introduce level zero definitions, or return level zero to full use. Definitions on a given level override definitions on any lower definition level. Levels higher than zero are largely for the internal use of LSQVAL.

LSQDEF(0, 5HBEGIN) introduces a new highest level of definition.

LSQDEF(*symb*, *lis*) defines the expression symbol *symb* to be the expression *lis*. This definition is made on the highest current definition level. It replaces any other definition of *symb* on the same level, and overrides any other definition of *symb* on a lower level. If the expression *lis* is simply a representation of an expression symbol *symb'*, then before the definition is made *lis* is replaced by the definition of *symb'*, if any, found on the highest non-zero level containing such a definition. An attempt to define an expression symbol to be itself will be ignored. An attempt to define an expression symbol to be an expression involving itself will be accepted and may cause an infinite loop (or loop through the limit level). Whether a loop occurs or not depends on how the expression symbol appears in its definition. If such an appearance is reached in the course of evaluation, a loop will occur.

The expression *lis* should be considered to be erased by this call to LSQDEF.

LSQDEF(*symb*, 0) "defines" the expression symbol *symb* to be undefined. This action is taken on the highest current level of definition. It replaces any definition of *symb* on the same level, and overrides any other definition of *symb* on a lower level. As long as the level on which this un-definition was made exists

and no new definition of *symb* is made on a higher level, the expression symbol *symb* acts as if it were totally undefined despite any old or new definitions made on level zero.

LSQDEF(0, 3HEND) discards the current highest level of definition and all definitions made thereon. The next definition level down becomes the current highest level of definition. It is an error to attempt to discard level zero in this manner.

Though it is not necessarily an error to do otherwise, the four two-argument calls to LSQDEF described above should normally be used in the following order:

```
      .
      .
      .
CALL LSQDEF(0, 5HBEGIN)
      .
      .
      .
CALL LSQDEF(symb1, lis1)
    any other substitution definitions
CALL LSQDEF(symbk, 0)
    any other "un-definitions"
CALL LSQVAL(lis)
    any other evaluations using these definitions
      .
      .
      .
CALL LSQDEF(0, 3HEND)
```

That is essentially the technique used within LSQVAL when it encounters a level zero definition with dummy arguments. The dummy arguments are defined to be the actual arguments and the definition is evaluated. Any dummy arguments which are not matched by actual arguments are left unmatched, causing any other existing definition of the unmatched dummies to take effect.

LSQDEF(*symb*, lis_1, lis_2) defines the expression symbol *symb* to be the expression lis_1 in which those expression symbols which appear on the list lis_2 are dummies for any actual arguments to which *symb* may be applied. This definition is made on definition level zero, and replaces any prior definition of *symb* on this level. The list lis_2 is of the form ($symb_1$, $symb_2$, ..., $symb_k$) for $k = 0, 1, 2, \ldots\infty$. The expression symbol $symb_i$ is used as a dummy for the i^{th} actual argument by calls to LSQDEF from LSQVAL as above. Thus the user may think of this call as

$$symb(symb_1,\ symb_2,\ \ldots,\ symb_k) \rightarrow lis_1$$

Though use of such a definition with fewer actual arguments than dummies raises the problems described above, having more actual arguments than dummy arguments yields reasonable results. Any residual actual arguments are evaluated and appended to the argument lists of all expression symbols in the evaluate of the definition, which lie on level 1 of the evaluate.

The expression lis_1 and the list lis_2 should be considered to

be erased by this call.

LSQDEF(*symb*, *lis*, 0) defines the expression symbol *symb* to be the expression *lis*. There are no dummy arguments to be considered. This definition is made on definition level zero, but is otherwise similar to calling LSQDEF(*symb*, *lis*). Again the user must consider the expression *lis* to be destroyed by this call.

LSQDEF(*symb*, 0, 0) revokes any prior definition of the expression symbol *symb* made on definition level zero and leaves *symb* undefined on this level.

LSQDEF(*symb*, lis_1, lis_2, 5HNØCUR) defines the expression symbol *symb* to be the expression lis_1 with dummy arguments as specified on the list lis_2. This call differs from the similar three argument call only in that when the definition expression is evaluated LSQDEF(*symb*, 0) will be called to make the expression symbol *symb* appear to be undefined. This permits *symb* to appear within the expression lis_1 without causing infinite loops.

LSQDEF(*symb*, *lis*, 0, 5HNØCUR) acts as the call above except that there are no dummy arguments involved.

LSQDEF(*symb*, *loc*, *int*) defines the expression symbol *symb* to be the result of a VISIT to the location *loc*, established in an ASSIGN statement. The integer quantity *int* determines the

handling of arguments and whether previously computed evaluates of the same function call may be used if available. If each time the code at location *loc* is applied to the same arguments, the same result is returned and no changes are made to variables, arrays, lists, expressions, etc. which might be used elsewhere, then the integer quantity *int* should be negative. Otherwise *int* should be positive. If the code at *loc* should be presented with unevaluated arguments, then *int* should be of magnitude 1. If the code should have evaluated arguments, then *int* should be of magnitude 2. Thus the four possible values for the integer quantity *int* are:

-2	no side effects, evaluate arguments
-1	no side effects, do not evaluate arguments
1	side effects, do not evaluate arguments
2	side effects, evaluate arguments

The user is again reminded that all unspecified values are reserved for future expansion of the system.

The evaluation code at location *loc* communicates with LSQVAL via the CØMMØN block LSQDMF, defined by

CØMMØN/LSQDMF/LSQDMF(15)

where LSQDMF(13) will contain a sequence reader which may be advanced

to the right to reach the first argument, to the right again to reach the second argument, and so forth. LSQDMF(14) will contain the expression symbol which invoked the VISIT. LSQDMF(15) will contain the number of arguments. An expression may be evaluated by storing the expression in LSQDMF(11) and paying a VISIT to LSQDMF(6). An integer one less than the level within the expression being evaluated on which the VISIT to *loc* was invoked may be found in LSQDMF(8). The limit level for evaluation will be an integer in LSQDMF(9).

A value *lis* should be returned to LSQVAL by a call to TERM(*lis*). The value must be a valid expression, but need not be a newly created expression. LSODES will be applied to *lis* to erase it if it was newly created.

To aid in writing code to be visited in an evaluation, the functions LSQDSF, LSQGAR, and LSQVVL are provided.

LSQDSF(*fun*, *symb*) returns the result of applying the FORTRAN callable function *fun*, which should be declared in an EXTERNAL statement, to the next expression, if any, to be reached by a right advance of LSQDMF(13). If the expression found is itself a constant expression, the result of calling LSQDSF will be a constant expression. Otherwise an expression will be formed by using the expression symbol *symb* as a function name and the expression found by the right advance of LSQDMF(13) as its argument. For example, calling the following subroutine would provide definitions of the arcsine and arccosine functions using the

expression symbols 6HARCSIN and 6HARCCØS respectively.

```
      SUBRØUTINE LSQASC
      EXTERNAL ASIN, ACØS
      ASSIGN 101 TØ LØC
      CALL LSQDEF(6HARCSIN, LØC, -2)
      ASSIGN 102 TØ LØC
      CALL LSQDEF(6HARCCØS, LØC, -2)
      RETURN
  101 CALL TERM(LSQDSF(ASIN, 6HARCSIN))
  102 CALL TERM(LSQDSF(ACØS, 6HARCCØS))
      END
```

LSQGAR(0) returns the next expression to be reached by a right advance of LSQDMF(13) if there is any next expression. Otherwise the value returned is zero.

LSQGAR(*symb*) returns the next expression to be reached by a right advance of LSQDMF(13) if there is any next expression. Otherwise an error abort is taken with the expression symbol *symb* provided as an error message.

LSQVVL(*lis*) returns a location to VISIT in order to evaluate the expression *lis*. The VISIT will save and restore LSQDMF(13) so that further scanning of arguments will be possible.

Consider the matter raised in §3.2 concerning the use of the expression symbol 6HEQUAL. as the translation of the equals sign on input via INLIST. The following subroutine will, when called, cause 6HEQUAL. to be defined as a means of creating definitions.

```
      SUBRØUTINE LSQIEQ
      ASSIGN 100 TØ LØC
      CALL LSQDEF(6HEQUAL., LØC, 1)
      RETURN
100   LA=LSQGAR(6HEQUAL.)
      NLA=LSQTYP(LA,NAM)
      IF((NLA.LT.10000).ØR.(NLA.GE.20000))
     *    CALL LSQERR(6HEQUAL.)
      SA=SEQRDR(LA)
      STU=SEQRDR(SEQLR(SA,N))
      SAC=SEQLR(STU,N)
      LTU=LIST(9)
101   SAR=SEQLR(STU,N)
      IF(N.NE.0)GØ TØ 102
      MLA=LSQTYP(SAR,MAM)
      IF(MLA.NE.10000)CALL LSQERR(6HEQUAL.)
      CALL MANY(LTU,MAM)
      GØ TØ 101
102   LB=LSQGAR(6HEQUAL.)
      CALL LSQDEF(NAM,LSQCPY(SEQRDR(LB)),LTU)
      CALL TERM(LB)
      END
```

Note that this limits EQUAL. to two useful arguments.

We may also wish to define the expression symbol 5HPRINT to print the value of its first argument with its second argument for a name. The following subroutine will, when called, so define 5HPRINT.

```
      SUBROUTINE LSQIPN
      CØMMØN AVSL, X(100)
      CØMMØN/LSQNTR/NTRYPT(10)
      CØMMØN/LSQARG/LARG(10)
      CØMMØN/LSQINT/INI
      CALL LSQPNT(NTRYPT(1))
      ASSIGN 100 TØ LØC
      CALL LSQDEF(5HPRINT,LØC,1)
      RETURN
  100 LA=LSQGAR(5HPRINT)
      CALL STRDIR(VISIT(LSQVVL(LA)),LARG(1))
      LB=LSQGAR(5HPRINT)
      IF(LSQTYP(LB,LARG(2)).NE.10000)CALL LSQERR(5HPRINT)
      CALL MANY(X(98),LARG(1))
      CALL VISIT(NTRYPT(1))
      CALL LSQUNM(X(98),LA)
      CALL TERM(LA)
      END
```

In both these examples, the definitions involve side effects and unevaluated arguments are used. Actually, when "unevaluated" arguments are requested, arguments evaluated with definition level

zero suspended are provided. Usually this simply causes actual arguments to be substituted for dummy arguments. Thus, if both the routines LSQIEQ and LSQIPN have been called, input lines of the form

```
PR(A..1)=PRINT(A..1,A..1)$$
LA=1+SIN(X)$$
PR(LA)$$
```

will generate output of the form

```
          LA = 1  +  SIN(X)
$
$END ØF EXPRESSIØN
```

rather than

```
          A..1 = 1  +  SIN(X)
$
$END ØF EXPRESSIØN
```

since PRINT will be invoked by the evaluation of PR with the actual argument LA substituted for the dummy A..1.

If the user should do all his expression printing by means of evaluation of an expression symbol such as SHPRINT, then he may use the entries to LSQVAL by means of VISITs within special print definitions.

LSQDEF(0, 4HSAVE) suspends all level zero definitions and makes it an error to introduce any level zero definitions.

LSQDEF(0, 6HUNSAVE) restores definition level zero to full use.

It is always proper to use these two calls (in pairs) within FORTRAN code used as a level zero definition, since in order to have reached the code, level zero must not have been suspended already. In all other cases the user must be careful not to call LSQDEF(0,4HSAVE) twice in a row.

EVALUE(lis, $symb_1$, $quan_1$, ..., $symb_k$, $quan_k$) for k = 1, 2, 3, 4, 5, 6, evaluates the expression lis with the expression symbols $symb_i$ set to the floating point constants $quan_i$. The evaluate is returned as the value of the function and also replaces the expression lis. Inasmuch as this is a destructive manipulation of an expression, calls to EVALUE should be avoided in new code.

HITENT(lis_1, $symb$, lis_2, lis_3) returns the result of adding a level one substitution of the expression lis_1 for the expression symbol $symb$ in the expression lis_2 to the expression lis_3 which is replaced by this result. LSQSBS is a better choice for new code. This substitution leaves parentheses around the uses of lis_1.

INSBST(lis_1, $symb$, lis_2, lis_3) returns the result of adding an all levels substitution of the expression lis_1 for the expression symbol $symb$ in the expression lis_2 to the expression lis_3 which is replaced by this result. LSQSBS is again a better choice for new code.

INSUBT(lis_1, $symb$, lis_2, lis_3, $symb_1$, $quan_1$, ..., $symb_k$, $quan_k$) for k = 1, 2, 3, 4, 5, returns the result of adding an all levels substitution of the expression lis_1 for the expression symbol $symb$ in the expression lis_2 to the expression lis_3, and then truncating on the expression symbols $symb_i$ to the powers given by the floating point quantities $quan_i$. The expression lis_3 is replaced by the

result. LSQSBS and LSQTRC should be used in new code.

INTENT(lis_1, $symb$, lis_2, lis_3) acts as HITENT except that substitution is done on all levels.

SBST(lis_1, $symb$, lis_2, lis_3) acts as INSBST except that substitution is restricted to level one.

SUBT(lis_1, $symb$, lis_2, lis_3, $symb_1$, $quan_1$, ..., $symb_k$, $quan_k$) for k = 1, 2, 3, 4, 5, acts as INSUBT except that substitution is restricted to level one.

Exercise

3.6.1. Assume that the INPUT and ØUTPUT files are actually a teletype. Write a program using the subroutines LSQIEQ and LSQIPN and such additional code as you may need to form a desk calculator for expressions. You would do well to use the three argument call to INLIST, so that LSQIEQ will have something to work on. Consider including a definition for the expression symbol STØP which will terminate execution. Also consider ways to allow the input expressions to begin in column 1.

3.7. Truncation Recapped

The function LSQTRC, mentioned in §2.8, truncates expressions using the concept of levels described for evalauation in §3.6.

LSQTRC(*lis*, *symb*, $quan_1$, $quan_2$, int_1, int_2) returns an expression derived from expression *lis* by retaining certain terms of the expression through level int_1, an integer quantity, and all terms on deeper levels. Whether a term is retained or not depends on whether it contains the expression symbol *symb* to a non-constant power or to a power between the floating point quantities $quan_1$ and $quan_2$.

If the integer quantity int_2 is 0, 2, 4, 6, 8, 10, 12, or 14, then terms containing the expression symbol *symb* to a constant power lying between $quan_1$ and $quan_2$ will be retained. If, on the other hand int_2 is 1, 3, 5, 7, 9, 11, 13, or 15, then terms containing *symb* to a constant power lying outside of the range between $quan_1$ and $quan_2$ inclusive. If int_2 is 0, 1, 4, 5, 8, 9, 12, or 13, the terms containing *symb* to non-constant powers will not be retained; while if int_2 is 2, 3, 6, 7, 10, 11, 14, or 15, terms with *symb* to non-constant powers will not be retained. If int_2 is 0, 1, 2, 3, 8, 9, 10, or 11, then terms containing *symb* to a constant power equal to the maximum of $quan_1$ and $quan_2$ will not be retained, while for int_2 equal to 4, 5, 6, 7, 12, 13, 14, or 15, they will be retained. If int_2 is 0, 1, 2, 3, 4, 5, 6, or 7, then terms containing *symb* to a constant power equal to the minimum of $quan_1$ and $quan_2$ will not be retained, while for int_2 equal to 8, 9, 10, 11, 12, 13, 14, or 15,

they will be retained. A term which does not contain the expression symbol *symb* (other than within parentheses or function arguments) is considered to contain *symb* to the constant power zero. Appearances of *symb* within parentheses or function arguments do not affect the decision as to whether a term should be retained or not. Appearances of *symb* with its own arguments are considered. Terms are scanned from left to right, and the first appearance of a factor which gives grounds for dropping the term will cause the term to be dropped.

LSQTRC deals in terms of the minimum and the maximum of the floating point quantities $quan_1$ and $quan_2$, so that they may be interchanged without changing the result.

As in §2.8, LSQTRC may be called with fewer than six arguments. This will cause default values to be used for $quan_1$, $quan_2$, int_1, and int_2 when the call fails to specify them. The arguments *lis* and *symb* must always be provided. The default values are as follows:

2 argument call $quan_1$=0, $quan_2$=0, int_1=1, int_2=12

3 argument call $quan_2$=0, int_1=1, int_2=12

4 argument call int_1=1, int_2=12

5 argument call int_2=12

Thus in these defaults, the expression *lis* will be truncated to constant powers within a closed interval.

The user may find it convenient to think of the truncation mode specified by int_2 in terms of the interior or exterior of an interval with or without the left and right endpoints. The mode is then the sum of four quantities:

interior	0 for the interior of the interval
	1 for the exterior of the interval
constant	0 for expression exponents not retained
	2 for expression exponents retained
right open	0 to exclude the right endpoint of the interval
	4 to include the right endpoint of the interval
left open	0 to exclude the left endpoint of the interval
	8 to include the left endpoint of the interval

Thus the default value of int_2, 12, specifies the interior with both endpoints and no non-constant exponents:

```
[--------------------]
```

while setting int_2 to 7 specifies the exterior with non-constant exponents and the right endpoint, but not the left:

```
----------)                    [----------
```

TRUNC(*lis*, *symb*, *quan*) is a truncation routine used in some existing code, which should not be used in new code. This routine replaces the expression *lis* with the result of discarding all first level terms which contain the expression symbol *symb* to a power greater than the floating point quantity *quan*, or to a non-constant power.

3.8. Further Means of Analysis of Lists and Expressions

There are many more facilities for examining lists and expressions than were covered in §2.10. The functions ADVLEL, ADVLER, ADVLNL, ADVLNR, ADVLWL, ADVLWR, ADVSEL, ADVSER, ADVSNL, ADVSNR, ADVSWL, ADVSWR, which we represent by "ADVαβγ", BØT, BREAK, GETCØE, INITRD, IRARDR, LCNTR, LØCT, LØFRDR, LPNTR, LRDPCP, LRDRØV, LVLRV1, LVLRVT, MADLFT, MADNBT, MADNTP, MADRGT, NAMTST, POWER, PEED, SEQSL, SEQSR, SØLVE, TØP, TRCAL, and TSTCØN may be so used.

BØT(*lis*) returns the bottom (rightmost) element of the list or expression *lis*, which is assumed to be non-empty. The element returned remains on the list.

TØP(*lis*) returns the top (leftmost) element of the list or expression *lis*, which is assumed to be non-empty. The element returned remains on the list.

If LA is a non-zero expression, then LSQMNL(TØP(LA)) and LSQMNL(BØT(LA)) are properly formed expressions consisting of the leftmost and rightmost terms of LA respectively.

LØCT(*lis*) returns the list or expression *lis*, provided *lis* is actually a list. Otherwise LØCT forces an error abort. This routine is used by subroutines to insure that arguments which are supposed to be lists actually are lists.

BREAK(lis_1, *symb*, lis_2, lis_3) adds to the expression lis_2 those terms of the expression lis_1 which contain the expression symbol *symb* on the first level, replacing lis_2 with this result. The remaining terms of lis_1 are added to the expression lis_3, which is changed to this new value. This routine should not be used in new code. Calls to LSQTRC are preferable.

GETCØE(*symb*, *quan*, lis_1, lis_2) adds to the expression lis_2 the coefficient of the expression symbol *symb* raised to the floating point constant power *quan* in the expression lis_1, replacing lis_2 with the result and returning it as the function value. This routine should not be used in new code.

NAMTST(*quan*) returns zero only if the quantity *quan* is a list or expression, -1 otherwise.

PØWER(*lis*, *symb*) returns the leftmost power to which the expression symbol *symb* is raised in the expression *lis*, considering only the first level. Zero is returned if *symb* is not found.

SØLVE(lis_1, *symb*, lis_2) assumes that the expression lis_1 is linear in the expression symbol *symb* and solves the equation $lis_1 = 0$ for *symb*. The solution is added to the expression lis_2 which is replaced by the sum. The sum is also returned as the function value. This routine should not be used in new code.

TRCAL(*lis*, $symb_1$, $symb_2$) finds the lowest power to which the expression symbol $symb_2$ is raised in those terms of the expression *lis* which contain the expression symbol $symb_1$. Then the expression *lis* is replaced by a truncated copy in which those terms that contain the expression symbol $symb_1$ to a power greater than that minimal power of $symb_2$ are discarded. This routine should not be used in new code.

TSTCØN(*lis*, *var*) returns zero if the expression *lis* is not a constant expression. If *lis* is a constant expression, the value returned is 1.0, and the variable or array element *var* is set to the equivalent FORTRAN constant value. LSQTYP should be used in new code.

SEQSL(*var*, *intvar*) returns the next list element other than a sublist encountered in left advances of the variable or array element *var* treated as a sequence reader, and by descents into any sublists encountered. The integer variable or array element *intvar* will be set to 1 if a list head stops the search, and -1 if a list element other than a sublist or list head is found. SEQLL and SEQLR may be applied to the sequence reader *var* after use by SEQSL, but the user should be aware that ascent back to higher level lists is not possible.

SEQSR(*var*, *intvar*) returns the next list element other than a sublist encountered in right advances of the variable or array element *var* treated as a sequence reader, and by descents into any sublists encountered. The integer variable or array element *intvar* will be set to 1 if a list head stops the search, and -1 if a list element other than a sublist or list head is found. The same caution as to subsequent use of SEQLL and SEQLR hold as for SEQSL.

As an example, consider the following code assuming LA to be an expression.

```
SA=SEQRDR(LA)
SCØ=SEQSR(SA,N)
```

If LA represents [[0]] then N will be 1; otherwise N will be -1, SCØ will be the coefficient of the leftmost term of LA, and SA will be a sequence reader for that term (not for LA) which is ready to advance to the right to the first factor.

MADLFT(*quan*) returns a quantity which may be used as a sequence reader for a right advance to the list element to the left of the list element which would be reached by a right advance of the quantity *quan* treated as a sequence reader. Thus the list element on which *quan* is sitting will be reached by a right advance of the result.

MADRGT(*quan*) returns a quantity which may be used as a sequence reader for a right advance to the list element to the right of the list element which would be reached by a right advance of the quantity *quan* treated as a sequence reader.

MADNBT(*lis*, *int*) returns a quantity which may be used for a right advance to the element of list or expression *lis* which is *int* elements to the left of the head of *lis*, where *int* is a non-zero positive integer quantity. If there are fewer than *int* elements in *lis*, then the search continues around the list, counting the head as an element each time it is encountered.

MADNTP(*lis*, *int*) returns a quantity which may be used for a right advance to the element of the list or expression *lis* which is *int* elements to the right of the head of *lis*, where *int* is a non-zero positive integer quantity.

The remaining routines discussed in this section are principally concerned with list readers, a more flexible tool of list scanning than sequence readers, in that a list reader may descend and ascend within a list and its sublists easily. A concept of list level is used in conjunction with list readers that is simply related to level within an expression. A level in an expression lies on two list levels: the one on which the terms of the expression are encountered, and the one on which the term coefficient and factors lie. Term coefficients on expression level n lie on list level $2n$-1.

LRDRØV(*lis*) returns a newly created list reader for the list or expression *lis*. A list reader is itself very similar to a list, and draws on the same available space list for a head and for elements to use to hold enough information to ascend from sublists into which it descends, as do lists and expressions.

IRARDR(*quan*) erases the list reader *quan*, and returns as it value the list level the reader had reached when erased. List readers should be erased when not needed so that their cells will be available for use in lists and other list readers.

ADV$\alpha\beta\gamma$(*quan*, *var*), where α is L or S, β is W, E, or N, and γ is L or R, returns the next list element reached by an advance of the list reader specified by the quantity *quan* in the direction γ (L for left, R for right) either restricted to the current list level (α = L) or descending into and rising from sublists when encountered (α = S), and considering the search satisfied either by a list head or a list element specified by β as follows: β is W for the next list element (whether or not a sublist), E for the next element which is not a sublist, and N for the next list element which is a sublist. The variable or array element *var* is set to -1.0 if the search is stopped by a list head, 0.0 otherwise.

INITRD(*quan*) returns the list reader which the quantity *quan* represents and advances that reader linearly to the head of the list or sublist on the list level being scanned.

LCNTR(*quan*) returns the list level to which the list reader specified by the quantity *quan* has descended. LCNTR may also be applied to a list, in which case the number of times that list appears as a sublist will be returned.

LØFRDR(*quan*) returns the list or sublist currently being scanned by the list reader specified by the quantity *quan*.

LPNTR(*quan*) returns a quantity which may be used as a sequence reader for a right advance to the list element to which the list reader specified by the quantity *quan* currently refers, i.e. the element to which it last advanced.

LRDRCP(*quan*) returns a copy of the list reader specified by the quantity *quan*. The copy will be in the same state as the original with respect to the elements it would reach on the next advance.

LVLRV1(*quan*) returns the list reader *quan* and, if a sublist is being scanned, makes the reader ascend one list level back to the point of descent.

LVLRVT(*quan*) returns the list reader *quan* and, if a sublist is being scanned, makes the reader ascend as many list levels as necessary to return to the point of descent on level zero.

REED(*quan*) returns the list element that the quantity *quan* currently references when treated either as a list reader or as a sequence reader. The element returned is the one reached on the last advance, the one on which *quan* sits.

The function LSQCMP described in §2.10 has some additional calls which are useful in the course of detailed analysis of an expression.

LSQCMP(lis_1, lis_2, 3HTER) returns zero if the terms lis_1 and lis_2 (i.e. sublists of expressions) are equal, -1 if lis_1 is lexicographically less than lis_2, +1 if lis_2 is lexicographically less than lis_1.

LSQCMP(var_1, var_2, 3HFAC) returns zero if the factors reached on the right advances of the sequence readers in the variables or array elements var_1 and var_2 are equal, -1 if the factor reached in right advances of var_1 is lexicographically less than that obtained from var_2, +1 otherwise. The next right advance of the sequence readers will bring them to the powers of the factors which are <u>not</u> considered in the comparison.

LSQCMP(lis_1, lis_2, 3HWØC) acts as the call with 3HTER in place of the third argument 3HWØC, except that both terms are considered to have coefficient 1., i.e. the coefficients are ignored in the comparison.

3.9. Attribute-Value Lists

Considerable use is made within SYMBØLANG of a SLIP mechanism known as the attribute-value list, which allows properties of lists to be declared. The attributes used internally in SYMBØLANG are 4HNAME, 7HNØRECUR, 6HSIMPLE, and 5HVALUE. These should be considered reserved to the package. To insure compatibility with future versions the user should not use as attributes any expression symbol of four or more characters.

The functions ITSVAL, MAKEDL, MTDLST, NAMEDL, and NEWVAL provide access to attribute-value lists.

ITSVAL(*quan*, *lis*) returns the value of the attribute *quan* for the list or expression *lis*. When a list is first created, all its attributes have the value zero. The quantity *quan* is not restricted, but use of list in this context is not recommended.

NEWVAL($quan_1$, $quan_2$, *lis*) gives the attribute $quan_1$ the value $quan_2$ on the list *lis*. The old value of the attribute is returned as the function value. For example for a new list LA

```
KA=NEWVAL(3HAGE,36,LA)
KB=NEWVAL(3HAGE,37,LA)
```

will leave KA set to zero, KB set to 36, and the attribute 3HAGE on the list LA with the value 37. Giving an attribute the value zero is as good as removing it.

Attributes are not elements of the list with which they are associated, and cannot be detected by use of sequence readers or list readers. Rather they lie on a separate list pointed to by an otherwise unused field in the head of the list with which they are associated. The exact format of this attribute value list should not be assumed by the programmer, so that it may be freely varied to improve the efficiency of future versions of SYMBOLANG.

NAMEDL(*lis*) returns the attribute-value list of the list or expression *lis*, zero if there is none.

MAKEDL(lis_1, lis_2) returns the list or expression lis_2, and makes the list lis_1 the attribute-value list of lis_2.

Thus LB may be made a very complete copy of list or expression LA by the following code.

```
LB=LSSCPY(LA)
LAV=NAMEDL(LA)
IF(LAV.NE.0) CALL MAKEDL(LSSCPY(LAV),LB)
```

MTDLST(*lis*) returns the list or expression *lis*, and empties its attribute-value list. Thus, every attribute will have the value zero after this call.

3.10. Another Example

As a final example of SYMBØLANG programming, we present the code of a preprocessor for SYMBØLANG written in SYMBØLANG. The bulk of the code is in the subroutine DEFS which defines the expression symbols 2HX. and 2HF. among others. Evaluation of X.(A) generates code for the representation of expression A as a list, while evaluation of F.(A) generates code to combine the expression symbols in the expression A using LSQADD for addition, LSQMEX for multiplication and LSQRAZ for exponentiation.

The user of this preprocessor accesses these and other definitions by presenting FORTRAN-like statements which begin with the symbol ≠ (the apostrophe on most keypunches). Lines beginning with ≠ in column 1 are treated as comments. Statements which do not begin with ≠ are transmitted to the file CØMPILE unchanged. Those which do are evaluated, and the generated expressions, if any, are transmitted as subroutine calls. INPUT lines are sent to the ØUTPUT file with line numbers and the generated code with the leading characters LSQ. A fragment of such output follows the program listing.

The user may define his own special functions by using the statement

$$\neq\mathrm{DEF.}(symb(symb_1,symb_2,\ldots),expression)$$

which defines *symb* with dummy arguments $symb_1$, etc., to be *expression*.

```
      PROGRAM XLSQ(INPUT,OUTPUT,COMPILE,TAPE1=COMPILE,TAPE5=INPUT,      XLSQ0002
     * TAPE6=OUTPUT)                                                     XLSQ0003
      CALL BEGIN                                                        XLSQ0004
      CALL MIDDLE                                                       XLSQ0005
      END                                                               XLSQ0006
      FUNCTION LSQERR(IA)                                               LSQE0002
      COMMON/TRAP/ITRAP,NOW,IWHAT                                       LSQE0003
      IWHAT=IA                                                          LSQE0004
      GO TO ITRAP                                                       LSQE0005
      END                                                               LSQE0006
      IDENT ICORE                                                       ICOR0002
      ENTRY ICORE                                                       ICOR0003
      VFD       42/0LICORE,18/1                                         ICOR0004
ICORE DATA      0                                                       ICOR0005
      MX6       0                                                       ICOR0006
      SA6       =SSTAT                                                  ICOR0007
      MEMORY    CM,STAT,RECALL                                          ICOR0008
      SA1       STAT                                                    ICOR0009
      SB3       30                                                      ICOR0010
      AX6       B3,X1                                                   ICOR0011
      EQ        ICORE                                                   ICOR0012
      END                                                               ICOR0013
      SUBROUTINE BEGIN                                                  BEGI0002
      COMMON/TRAP/ITRAP,NOW,IWHAT                                       BEGI0003
      COMMON AVSL,X(100),Y(5000)                                        BEGI0004
      ASSIGN 100 TO ITRAP                                               BEGI0005
      L=ICORE(DUM)-MADOV(Y)-3                                           BEGI0006
      L=L/2*2                                                           BEGI0007
      CALL INITAS(Y,L)                                                  BEGI0008
      CALL DEFS                                                         BEGI0009
      CALL LSQPNT(0,5HNOPRE)                                            BEGI0010
      CALL LSQPNT(0,5HNOSUB)                                            BEGI0011
      CALL LSQPNT(0,5HFLOAT)                                            BEGI0012
      CALL LSQOUT(5HCFLAG,6,1H*)                                        BEGI0013
  100 RETURN                                                            BEGI0014
      END                                                               BEGI0015
      SUBROUTINE DEFS                                                   DEFS0002
      COMMON AVSL,X(100)                                                DEFS0003
      COMMON/TRAP/ITRAP,NOW,IWHAT                                       DEFS0004
      COMMON/TEMP/LTEMP,LDFS,NTEMP,LINT,NINT                            DEFS0005
      COMMON/LSQNTR/NTRYPT(10)                                          DEFS0006
      COMMON/LSQARG/LARG(10)                                            DEFS0007
      COMMON/LSQINT/INI                                                 DEFS0008
      DIMENSION ICBU(10),NAM(1),ISBU(2)                                 DEFS0009
C                                                                       DEFS0010
C     TRAPS FOR LSQPNT                                                  DEFS0011
C                                                                       DEFS0012
      ASSIGN 1010 TO LOC                                                DEFS0013
      CALL LSQPNT(INI,4HFIX.,LOC)                                       DEFS0014
      ASSIGN 1020 TO LOC                                                DEFS0015
      CALL LSQPNT(INI,2HS.,LOC)                                         DEFS0016
      ASSIGN 1030 TO LOC                                                DEFS0017
      CALL LSQPNT(INI,4HCAT,,LOC)                                       DEFS0018
C                                                                       DEFS0019
C     DEFINITIONS                                                       DEFS0020
C                                                                       DEFS0021
```

```
2000      CONTINUE
          ASSIGN 2010 TO LOC
          CALL LSQDEF(4HDEF,,LOC,1)
          ASSIGN 2020 TO LOC
          CALL LSQDEF(2HX,,LOC,-1)
          ASSIGN 3010 TO LOC
          ASSIGN 3011 TO IPX
          ASSIGN 3012 TO IPT
          CALL LSQDEF(2HF,,LOC,-2)
          CALL LSQDEF(3HF,,,LOC,-1)
          ASSIGN 3030 TO LOC
          CALL LSQDEF(2HT,,LOC,2)
          CALL LSQDEF(3HT,,,LOC,1)
          ASSIGN 3040 TO LOC
          CALL LSQDEF(3HRT,,LOC,2)
          ASSIGN 3050 TO LOC
          CALL LSQDEF(2HR,,LOC,2)
          ASSIGN 3060 TO LOC
          CALL LSQDEF(2HI,,LOC,2)
          RETURN
1010      SA=SEQLR(LARG(6),N)
          IF(N.NE.0)CALL LSQERR(4HFIX,)
          IF(LSQTYP(SA,COE).NE.0)CALL LSQERR(4HFIX,)
          LC=COE
          ENCODE(20,1011,ICBU)LC
1011      FORMAT(I20)
          CALL LSQOUT(0,LANORM(ICBU(1)),0,LANORM(ICBU(2)))
          CALL TERM(0,)
1020      SA=SEQLR(LARG(6),N)
          IF(N.NE.0)CALL LSQERR(2HS,)
          IF(LSQTYP(SA,NAM).NE.10000)CALL LSQERR(2HS,)
          DECODE(10,1021,NAM)ICBU
1021      FORMAT(10A1)
          L=0
          DO 1022 J=1,10
          IF(ICBU(J).NE.1H )L=L+1
1022      CONTINUE
          IF(L.EQ.0)CALL LSQERR(2HS,)
          LL=L+3
          ENCODE(LL,1023,ISBU)L,(ICBU(I),I=1,L)
1023      FORMAT(I2,1HH,10A1)
          CALL TERM(LSQOUT(LL,ISBU))
1030      CALL STRDIR(SEQLR(LARG(6),N),LARG(1))
          IF(N.NE.0)CALL TERM(0,)
          CALL NEWBOT(LARG(6),X(100))
          CALL VISIT(NTRYPT(3))
          CALL LSQUNM(X(100),LARG(6))
          GO TO 1030
2010      LA=LSQGAR(4HDEF,)
          LB=LSQGAR(0)
          NLA=LSQTYP(LA,NAM)
          IF(NLA.GE.20000.OR.NLA.LT.10000)CALL LSQERR(4HDEF,)
          LTV=0
          IF(LB.EQ.0.OR.NLA.EQ.10000)GO TO 2111
          MLA=NLA-10000
          LTV=LIST(9)
```

```
      SA=SEQRDR(TOP(LA))
      SCO=SEQLR(SA,N)
      DO 2112 J=1,MLA
      KLA=LSQTYP(SEQLR(SA,N),NEM)
      IF(KLA,NE,10000)CALL LSQERR(4HDEF,)
 2112 CALL NEWBOT(NEM,LTV)
 2111 CALL LSQDEF(NAM,LB,LTV)
      NOW=1
      CALL TERM(LIST(9))
C     CODE FOR X,(A)
C     TO BE USED WITHOUT EVALUATION, NO SIDE
C
C     RETURNS THE INTERNAL REP OF A EXCEPT IF A=USE,(B) OR A=Q,(R)
C     IN THE FIRST CASE, THE VALUE OF B IS RETURNED, IN THE SECOND,
C     THE INTERNAL REPRESENTATION
C
 2020 LA=LSQGAR(2HX,)
      NLA=LSQTYP(LA,NAM)
      IF(NLA,NE,10001)GO TO 2021
      SA=SEQRDR(TOP(LA))
      SCO=SEQLR(SA,N)
      IF(NAM,EQ,4HUSE,)CALL TERM(VISIT(LSQVVL(SEQLR(SA,N))))
      IF(NAM,NE,2HQ,)GO TO 2021
      CALL STRDIR(SEQLR(SA,N),LA)
 2021 SA=SEQRDR(LA)
      SAT=SEQLR(SA,N)
      LUM=LSQMNL(LSQMNL(1,))
      IF(N,EQ,0)GO TO 2023
      CALL MANY(TOP(LUM),LSQMNL(LSQMNL(1,,LSQMNL(LSQMNL(9,)),4HFIX,,
     *1,)),4HLIST,1,)
      CALL TERM(LUM)
 2023 SB=SEQRDR(SAT)
      LUMT=LSQMNL(LSQMNL(1,,LSQMNL(LSQMNL(SEQLR(SB,N)))))
 2025 SAX=SEQLR(SB,N)
      IF(N)2024,2026,2027
 2026 LAR=LSQMNL(LSQMNL(1,,SAX,2HX,,1,))
      CALL MANY(X(100),LUM,LUMT,LNKR(SA),LNKR(SB),LAR)
      CALL STRDIR(VISIT(LSQVVL(LAR)),LBR)
      CALL LSQUNM(X(100),LUM,LUMT,SA,SB,LAR)
      CALL LSQDES(LAR)
      CALL NEWBOT(LBR,BOT(LUMT))
      GO TO 2025
 2024 CALL NEWBOT(LSQMNL(LSQMNL(1,,LSQMNL(LSQMNL(1,,SAX,1,)),2HS,,1,
     *)),BOT(LUMT))
      SAX=SEQLR(SB,N)
      IF(N,EQ,0)GO TO 2026
      CALL NEWBOT(LSQMNL(LSQMNL(SAX)),BOT(LUMT))
      GO TO 2025
 2027 CALL MANY(BOT(LUMT),6HLSQMNL,1,)
      CALL NEWBOT(LUMT,BOT(LUM))
      SAT=SEQLR(SA,N)
      IF(N,EQ,0)GO TO 2023
      CALL MANY(BOT(LUM),6HLSQMNL,1,)
      CALL TERM(LUM)
C
C     F,
```

```
C
 3010     LA=LSQGAR(2HF.)
          CALL TERM(VISIT(IPX))
C
C         PROCESS F.(EXPRESSION)
C
 3011     SA=SEQRDR(LA)
          NT=0
 3013     SAR=SEQLR(SA,N)
          IF(N.NE.0)GO TO 3014
          CALL MANY(X(100),NT,LNKR(SA))
          V=VISIT(IPT)
          CALL LSQUNM(X(100),NT,SA)
          CALL NEWBOT(V,X(100))
          NT=NT+1
          GO TO 3013
 3014     IF(NT.EQ.0)CALL TERM(LSQMNL(LSQMNL(1.,LSQMNL(LSQMNL(1.,
     *    LSQMNL(LSQMNL(9.)),4HFIX.,1.)),4HLIST,1.)))
          LNAM=6HLSQADD
 3024     IF(NT.EQ.1) CALL TERM(POPBOT(X(100)))
          NT=NT-1
          LAN=LSQMNL(1.)
          LUN=LSQMNL(LSQMNL(1.,BOT(X(100)),3HT,,,1.))
          CALL MANY(X(100),LNAM,NT,LUN,LAN)
          V=VISIT(LSQVVL(LUN))
          CALL LSQUNM(X(100),LNAM,NT,LUN,LAN)
          CALL LSQDES(POPBOT(X(100)))
          CALL NEWBOT(V,LAN)
          CALL LSQDES(LUN)
 3015     NT=NT-1
          LUN=LSQMNL(LSQMNL(1.,BOT(X(100)),3HT,,,1.))
          CALL MANY(X(100),LNAM,NT,LUN,LAN)
          V=VISIT(LSQVVL(LUN))
          CALL LSQUNM(X(100),LNAM,NT,LUN,LAN)
          CALL LSQDES(POPBOT(X(100)))
          CALL MANY(LAN,V,LNAM,1.)
          CALL LSQDES(LUN)
          IF(NT.EQ.0)CALL TERM(LSQMNL(LAN))
          CALL MANY(X(100),LNAM,NT,LSQMNL(LSQMNL(1.,LSQMNL(LAN),3HT,,,1.
     *    )))
          V=VISIT(LSQVVL(BOT(X(100))))
          CALL LSQUNM(X(100),LNAM,NT,LUN)
          LAN=LSQMNL(1.,V)
          CALL LSQDES(LUN)
          GO TO 3015
C
C         PROCESS F.(TERM)
C
 3012     SAR=SEQRDR(SAR)
          SATC=SEQLR(SAR,N)
          CALL NEWBOT(LSQMNL(LSQMNL(1.,LSQMNL(LSQMNL(1.,LSQMNL(LSQMNL(
     *    SATC)),6HLSQMNL,1.)),6HLSQMNL,1.)),X(100))
          NT=1
          IF(SATC.NE.1)GO TO 3016
          SARA=SAR
          SARR=SEQLR(SARA,N)
```

```
      IF(N.GT.0)GO TO 3017
      CALL LSQDES(POPBOT(X(100)))
      NT=0
 3016 SARA=SAR
      IF(LSQGNF(SAR,NAM,NAR).GT.0)GO TO 3017
      NT=NT+1
      IF(NAM.EQ.3H1.*)GO TO 3019
      LUN=LSQMNL(1.)
 3018 CALL NEWBOT(SEQLR(SARA,N),LUN)
      IF(N.EQ.0)GO TO 3018
      CALL NEWBOT(LSQMNL(MANY(LUN,1.)),X(100))
 3022 SPOW=SEQLR(SAR,N)
      IF(N.GE.0)GO TO 3020
      IF(SPOW.NE.1.)GO TO 3021
      GO TO 3016
 3019 IF(NAR.NE.1)CALL LSQERR(2HF.)
      LUN=LSQMNL(LSQMNL(1.,SEQLR(SARA,N),3HF..,1.))
      CALL MANY(X(100),LNKR(SAR),NT,LUN)
      V=VISIT(LSQVVL(LUN))
      CALL LSQUNM(X(100),SAR,NT,LUN)
      CALL NEWBOT(V,X(100))
      CALL LSQDES(LUN)
      GO TO 3022
 3021 LUN=LSQMNL(LSQMNL(1.,LSQMNL(LSQMNL(1.,LSQMNL(LSQMNL(SPOW)),
     * 6HLSQMNL,1.)),6HLSQMNL,1.))
 3023 LRUN=LSQMNL(LSQMNL(1.,LSQMNL(LSQMNL(1.,POPBOT(X(100)),3HT..,
     * 1.)),LSQMNL(LSQMNL(1.,LUN,3HT..,1.)),6HLSQRAZ,1.))
      CALL MANY(X(100),LNKR(SAR),NT,LRUN)
      V=VISIT(LSQVVL(LRUN))
      CALL LSQUNM(X(100),SAR,NT,LRUN)
      CALL NEWBOT(V,X(100))
      CALL LSQDES(LRUN)
      GO TO 3016
 3020 LUN=LSQMNL(LSQMNL(1.,SPOW,3HF..,1.))
      CALL MANY(X(100),LNKR(SAR),NT,LUN)
      V=VISIT(LSQVVL(LUN))
      CALL LSQUNM(X(100),SAR,NT,LUNA)
      LUN=LSQCPY(SEQRDR(V))
      CALL LSQDES(LUNA)
      CALL LSQDES(V)
      GO TO 3023
 3017 LNAM=6HLSQMEX
      GO TO 3024
C
C     T.
C
 3030 LA=LSQGAR(2HT.)
      SA=SEQRDR(LTEMP)
 3032 SAN=SEQLR(SA,N)
      IF(N.GT.0)GO TO 3031
      IF(LSQCMP(SEQLR(SA,N),LA,3HEXP).NE.0)GO TO 3032
      CALL TERM(LSQMNL(LSQMNL(1.,SAN,1.)))
 3031 NLA=LSQTYP(LA,NAM)
      IF(NLA.NE.10000)GO TO 3033
      SA=SEQRDR(LDES)
 3037 SAN=SEQLR(SA,N)
```

```
      IF(N.GT.0)GO TO 3036
      IF(LSQCNM(SAN,NAM).EQ.0)CALL TERM(LA)
      GO TO 3037
 3036 CALL NEWBOT(NAM,LDES)
      CALL TERM(LA)
 3033 NTEMP=NTEMP+1
      ENCODE(6,3034,NAM)NTEMP
 3034 FORMAT(3HLSQ,O3)
      CALL MANY(LTEMP,NAM,LA)
      LA=LSQMNL(LSQMNL(1.,NAM,1.))
      GO TO 3036
C
C     RT.
C
 3040 IRM=0
      LA=LSQGAR(3HRT.)
 3047 NLA=LSQTYP(LA,NAM)
      IF(NLA.NE.10000)GO TO 3043
      SA=SEQRDR(LDES)
      MDES=LIST(9)
 3045 SAN=SEQLR(SA,N)
      IF(N.GT.0)GO TO 3044
      IF(LSQCNM(NAM,SAN).EQ.0) GO TO 3046
      CALL NEWBOT(SAN,MDES)
      GO TO 3045
 3046 NTEMP=NTEMP+1
      ENCODE(6,3034,NAM)NTEMP
      CALL MANY(LTEMP,NAM,LA)
      CALL NEWBOT(NAM,MDES)
      GO TO 3045
 3044 CALL LSQDES(LDES)
      LDES=MDES
 3043 SA=SEQRDR(LTEMP)
 3042 SAN=SEQLR(SA,N)
      IF(N.GT.0)GO TO 3041
      IF(LSQCMP(SEQLR(SA,N),LA,3HEXP).NE.0)GO TO 3042
      CALL TERM(LA)
 3041 IF(IRM.NE.0)CALL TERM(LA)
      NTEMP=NTEMP+1
      ENCODE(6,3034,NAM) NTEMP
      CALL MANY(LTEMP,NAM,LA)
      GO TO 3036
C
C     R.
C
 3050 IRM=1
      LA=LSQGAR(2HR.)
      GO TO 3047
C
C     I.
C
 3060 LA=LSQGAR(2HI.)
      NLA=LSQTYP(LA,TYP)
      IF(NLA.EQ.0)CALL TERM(LSQMNL(LSQMNL(1.,LA,4HFIX.,1.)))
      SA=SEQRDR(LINT)
 3062 SAN=SEQLR(SA,N)
```

```
      IF(N.GT.0)GO TO 3061
      IF(LSQCMP(SEQLR(SA,N),LA,3HEXP).NE.0)GO TO 3062
      CALL TERM(LSQMNL(LSQMNL(1.,SAN,1.)))
3061  NTEMP=NTEMP+1
      ENCODE(6,3034,NAM)NTEMP
      CALL MANY(LINT,NAM,LA)
      CALL TERM(LSQMNL(LSQMNL(1.,NAM,1.)))
      END
      SUBROUTINE MIDDLE
      DIMENSION INBUF(90),ISB(5),OSB(8),M(1)
      COMMON/TRAP/ITRAP,NOW,IWHAT
      COMMON/TEMP/LTEMP,LDFS,NTEMP,LINT,NINT
      COMMON/PAGE/LINE,IPAGE
      DATA IERR/0/,IOFF/0/
      ASSIGN 999 TO ITRAP
1     READ(5,100)INBUF
100   FORMAT(90R1)
      IF(ENDFILE 5)2,3
2     IF(IERR.NE.0) GO TO 21
      CALL EXIT
21    ENCODE(80,210,OSB)IERR
210   FORMAT(4H ***,I6,* LSQ ERRORS*)
      CALL ABORT(OSB,6L*CXIT.)
3     CALL LPRINT(INBUF,0)
      IF(IOFF.NE.0) GO TO 5
      IF(INBUF(1).NE.1R≠) GO TO 4
      ENCODE(10,100,M)(INBUF(I),I=2,11)
      IF(M.EQ.4HLAST) GO TO 2
      IF(M.EQ.3HOFF) GO TO 6
      GO TO 1
6     IOFF=1
      GO TO 1
5     IF(INBUF(1).NE.1R≠) GO TO 50
      ENCODE(10,100,M)(INBUF(I),I=2,11)
      IF(M.NE.2HON) GO TO 50
      IOFF=0
      GO TO 1
50    WRITE(1,100) INBUF
      GO TO 1
4     CALL SETRAY(ISB,5,1R )
      IF((INBUF(1).NE.1R ).AND.(INBUF(1).LT.1R0.OR.INBUF(1).GT.1R9))
     * GO TO 50
      IF(INBUF(6).NE.1R ) GO TO 50
      M=5
      DO 41 J=1,5
      LU=6-J
      IF(INBUF(LU).EQ.1R ) GO TO 41
      IF(INBUF(LU).LT.1R0.OR.INBUF(LU).GT.1R9) GOTO 50
      ISB(M)=INBUF(LU)
      M=M-1
41    CONTINUE
      ISTATN=10H
      IF(M.LT.5)ENCODE(5,100,ISTATN)ISB
      ISEND=0
      DO 77 J=7,72
      IF(INBUF(J).EQ.1R ) GO TO 77
```

```
      IF(INBUF(J).NE.1R≠) GO TO 50
      M=J
      GO TO 78
77    CONTINUE
      GO TO 50
78    INBUF(M)=1R
      LIS=LIST(9)
79    ENCODE(72,101,OSB)(INBUF(I),I=7,72)
101   FORMAT(6X,72R1)
      LYS=LIST(9)
      DO 80 J=1,8
80    CALL NEWBOT(OSB(J),LYS)
      CALL NEWBOT(LYS,LIS)
      INBUF(1)=1RC
      INBUF(2)=1R
      INBUF(3)=1RL
      INBUF(4)=1RS
      INBUF(5)=1RQ
      WRITE(1,100)INBUF
      READ(5,100)INBUF
      IF(ENDFILE 5)81,82
81    ISEND=1
      GO TO 83
82    IF(INBUF(6).EQ.1R )GO TO 83
      IF(INBUF(1).NE.1R )GO TO 83
      CALL LPRINT(INBUF,0)
      GO TO 79
83    NOW=0
      NTEMP=0
      LTEMP=LIST(9)
      LDES=LIST(9)
      NINT=0
      LINT=LIST(9)
      LAM=INLIST(LAM,LIS,0)
      CALL LSQDES(LIS)
912   IF(LISTMT(LINT).EQ.0)GO TO 911
      SAN=POPTOP(LINT)
      LYS=LIST(9)
      CALL LSQOUT(4HLIST,LYS,6HMARGIN,1,0,ISTATN,6HMARGIN,7,0,SAN)
      CALL LSQOUT(3,3H = )
      ISTATN=10H
      SAN=POPTOP(LINT)
      CALL LSQPNT(SAN,1)
      CALL LSQOUT(5HFLUSH)
      CALL LSQDES(SAN)
      CALL LOUT(LYS)
      GO TO 912
911   IF(LISTMT(LTEMP).EQ.0)GO TO 910
      SAN=POPTOP(LTEMP)
      LYS=LIST(9)
      CALL LSQOUT(4HLIST,LYS,6HMARGIN,1,0,ISTATN,6HMARGIN,7,12,
     * 12HCALL STRDIR()
      ISTATN=10H
      SAM=POPTOP(LTEMP)
      CALL LSQPNT(SAM,1)
      CALL LSQOUT(1,1H,,0,SAN,1,1H),5HFLUSH)
```

```
      CALL LOUT(LYS)                                                    MIDD0106
      GO TO 911                                                         MIDD0107
910   IF(NOW.EQ.0)GO TO 920                                             MIDD0108
      CALL LSQDES(LAM)                                                  MIDD0109
930   IF(LISTMT(LDES).EQ.0)GO TO 931                                    MIDD0110
      LYS=LIST(9)                                                       MIDD0111
      CALL LSQOUT(4HLIST,LYS,6HMARGIN,1,0,ISTATN,6HMARGIN,7,12,         MIDD0112
     * 12HCALL LSQDES(,0,POPBOT(LDES),1,1H),5HFLUSH)                    MIDD0113
      CALL LOUT(LYS)                                                    MIDD0114
      GO TO 930                                                         MIDD0115
931   CALL LSQDES(LTEMP)                                                MIDD0116
      CALL LSQDES(LINT)                                                 MIDD0117
      CALL LSQDES(LDES)                                                 MIDD0118
903   IF(ISEND.NE.0)2,3                                                 MIDD0119
920   ASSIGN 930 TO LAC                                                 MIDD0120
      IF(LSQTYP(LAM,NAM)-20000)922,923,923                              MIDD0121
922   IF(NAM.NE.3HDO.)GO TO 923                                         MIDD0122
      ASSIGN 924 TO LAC                                                 MIDD0123
      LUM=LAM                                                           MIDD0124
      SAT=TOP(LAM)                                                      MIDD0125
      SAR=SEQRDR(SAT)                                                   MIDD0126
      SAC=SEQLR(SAR,N)                                                  MIDD0127
924   CALL STRDIR(SEQLR(SAR,N),LAM)                                     MIDD0128
      IF(N.EQ.0)GO TO 923                                               MIDD0129
      CALL LSQDES(LUM)                                                  MIDD0130
      GO TO 930                                                         MIDD0131
923   LYS=LIST(9)                                                       MIDD0132
      CALL LSQOUT(4HLIST,LYS,6HMARGIN,1,0,ISTATN,6HMARGIN,7)            MIDD0133
      ISTATN=10H                                                        MIDD0134
      IF(LSQTYP(LAM,NAM)-20000)925,926,926                              MIDD0135
925   IF(NAM.NE.4HCAT.)GO TO 926                                        MIDD0136
927   CALL LSQPNT(LAM,1)                                                MIDD0137
      CALL LSQOUT(5HFLUSH)                                              MIDD0138
      CALL LSQDES(LAM)                                                  MIDD0139
      CALL LOUT(LYS)                                                    MIDD0140
      GO TO LAC,(930,924)                                               MIDD0141
926   CALL LSQOUT(5,5HCALL )                                            MIDD0142
      GO TO 927                                                         MIDD0143
999   IERR=IERR+1                                                       MIDD0144
      ISR(1)=10H *** ERROR                                              MIDD0145
      ISR(2)=10H***                                                     MIDD0146
      ISB(3)=IWHAT                                                      MIDD0147
      CALL LPRINT(ISB,3)                                                MIDD0148
      IF(IWHAT.NE.6HNUCELL) GO TO 903                                   MIDD0149
      CALL REMARK(28L *** MEMORY OVERFLOW ***                           MIDD0150
      GO TO 21                                                          MIDD0151
      END                                                               MIDD0152
      FUNCTION LPRINT(LBUF,IC)                                          LPRI0002
      DIMENSION LBUF(90)                                                LPRI0003
      COMMON/PAGE/LINE,IPAGE                                            LPRI0004
      DATA LINE/55/,IPAGE/0/,LPP/55/                                    LPRI0005
      DATA ISLIN/0/                                                     LPRI0006
      LINE=LINE+1                                                       LPRI0007
      IF(LINE.LE.LPP) GO TO 200                                         LPRI0008
      IPAGE=IPAGE+1                                                     LPRI0009
      CALL DATE(XD)                                                     LPRI0010
```

```
      CALL HOUR(XT)                                                     LPRI0011
      WRITE(6,101)XT,XD,IPAGE                                           LPRI0012
101   FORMAT(*1          LSQ 1.0 SYMBOLANG PREPROCESSOR          TIME LPRI0013
     *-*,A10,*          DATE -*,A10,*          PAGE -*,I6,/)             LPRI0014
      LINE=3                                                            LPRI0015
200   IF(IC.NE.0) GO TO 300                                             LPRI0016
      ISLIN=ISLIN+1                                                     LPRI0017
      WRITE(6,201)ISLIN,LBUF                                            LPRI0018
201   FORMAT(1X,I8,1X,90R1)                                             LPRI0019
      RETURN                                                            LPRI0020
300   WRITE(6,301)(LBUF(J),J=1,IC)                                      LPRI0021
301   FORMAT(6X,*LSQ *,8A10)                                            LPRI0022
      RETURN                                                            LPRI0023
      END                                                               LPRI0024
      FUNCTION LOUT(LYS)                                                LOUT0002
      DIMENSION IOBU(8)                                                 LOUT0003
5     IF(LISTMT(LYS).EQ.0)GO TO 1                                       LOUT0004
      SAM=POPTOP(LYS)                                                   LOUT0005
      L=1                                                               LOUT0006
      CALL SETRAY(IOBU,8,10H          )                                 LOUT0007
2     IF(LISTMT(SAM).EQ.0)GO TO 3                                       LOUT0008
      CALL STRDIR(POPTOP(SAM),IOBU(L))                                  LOUT0009
      L=L+1                                                             LOUT0010
      GO TO 2                                                           LOUT0011
3     CALL LPRINT(IOBU,8)                                               LOUT0012
      WRITE(1,4)IOBU                                                    LOUT0013
4     FORMAT(8A10)                                                      LOUT0014
      CALL LSQDES(SAM)                                                  LOUT0015
      GO TO 5                                                           LOUT0016
1     CALL LSQDES(LYS)                                                  LOUT0017
      RETURN                                                            LOUT0018
      END                                                               LOUT0019
```

```
  1 ≠
  2 ≠          SOURCE CARDS ARE SCANNED FOR AN ≠ ON THE FIRST CARD OF A
  3 ≠          STATEMENT.  IF NOT FOUND THE STATEMENT IS TRANSFERED TO THE
  4 ≠          COMPILE FILE.
  5 ≠
  6 ≠          IF AN ≠ IS FOUND BETWEEN COLUMNS 7 AND MI THE SOURCE
  7 ≠          STATEMENT IS OUTPUT AS A COMMENT AND THE EVALUATE OF THE
  8 ≠          SOURCE STATEMENT IS OUTPUT AFTER  ≠CALL≠.
  9 ≠
 10 ≠          THE NEXT CARD DEFINES THE EFFECT OF =
 11            ≠DEF.(A...1=A...2,STRDIR(A...2,R,(A...1)))
 12 ≠          NOW FOR A PROGRAM
 13 ≠
 14            PROGRAM TEST(INPUT,OUTPUT)
 15            DIMENSION SPACE(5000)
 16            CALL INITAS(SPACE,5000)
 17 C          ADD A COLUMN OF EXPRESSIONS
 18            ≠LA=X.(0)
LSQ        CALL STRDIR(LIST(9),LA)
 19  1         LB=INLIST(LB,5LINPUT)
 20            IF(LISTMT(LB).EQ.0)GO TO 2
 21            ≠LSQPNT(LB,S.(LB))
LSQ        CALL LSQPNT(LB, 2HLB)
 22            ≠LA=F.(LA+LB)
LSQ        CALL STRDIR(LA,LSQ001)
LSQ        CALL STRDIR(LSQADD(LB,LA),LA)
LSQ        CALL LSQDES(LSQ001)
LSQ        CALL LSQDES(LB)
 23            GO TO 1
 24  2         ≠LSQPNT(LA,S.(LA))
LSQ 2      CALL LSQPNT(LA, 2HLA)
 25
 26 C          NOW LETS TAKE THE SUM OF SOME POWERS
 27            ≠LA=F.(LA+LA**2+LA**3+LA**4)
LSQ        CALL STRDIR(LSQMNL(LSQMNL(2.)),LSQ001)
LSQ        CALL STRDIR(LSQMNL(LSQMNL(3.)),LSQ002)
LSQ        CALL STRDIR(LSQMNL(LSQMNL(4.)),LSQ003)
LSQ        CALL STRDIR(LSQRAZ(LA,LSQ003),LSQ004)
LSQ        CALL STRDIR(LSQRAZ(LA,LSQ002),LSQ005)
LSQ        CALL STRDIR(LSQADD(LSQ004,LSQ005),LSQ006)
LSQ        CALL STRDIR(LSQRAZ(LA,LSQ001),LSQ007)
LSQ        CALL STRDIR(LSQADD(LSQ006,LSQ007),LSQ010)
LSQ        CALL STRDIR(LA,LSQ011)
LSQ        CALL STRDIR(LSQADD(LSQ010,LA),LA)
LSQ        CALL LSQDES(LSQ010)
LSQ        CALL LSQDES(LSQ007)
LSQ        CALL LSQDES(LSQ006)
LSQ        CALL LSQDES(LSQ005)
LSQ        CALL LSQDES(LSQ004)
LSQ        CALL LSQDES(LSQ003)
LSQ        CALL LSQDES(LSQ002)
LSQ        CALL LSQDES(LSQ001)
LSQ        CALL LSQDES(LSQ011)
```

4. REFERENCES

[1] Control Data 6400/6500/6600 Computer Systems SCOPE 3.1.2 Reference Manual, publication number 60189400, Control Data Corporation, Minneapolis, Minn., 1968.

[2] Control Data 6400/6500/6600 Computer Systems FØRTRAN Reference Manual, publication number 60174900, Control Data Corporation, Minneapolis, Minn., 1969.

[3] Bernstein, H.J., "The Program SLIP - A Program Providing Access to the Routines of the SYMBOLANG and SLIP Packages - A Preliminary Report", document in deck SLIP on NYU supported writeups tape T599, AEC Computing & Applied Math. Center, New York Univ., New York, June 1969.

[4] Bosworth, C.R., "SELECT", the Aerospace Corporation, November 1968. (Available in part in the deck SELECT on NYU supported writeups tape T599).

[5] Engeli, M.E., "Achievements and Problems in Formula Manipulation", Proc. IFIP Congress 68, invited papers, North-Holland Publishing Co., 1968, pp. 79-84.

[6] Engelman, C., "MATHLAB 68", Proc. IFIP Congress 68, software I, North-Holland Publishing Co., 1968, pp. B91-B95.

[7] Harrison, M.C. & Schwartz, J.T., "SHARER, a Time Sharing System for the CDC 6600", Comm. ACM, vol. 10, no. 10, October 1967, pp. 659-665.

[8] Lapidus, A. & Goldstein, M., "Some Experiments in Algebraic Manipulation by Computer", Comm. ACM, vol. 8, no. 8, August 1965, pp. 501-508.

[9] Lapidus, A., Goldstein, M., & Hoffberg, S.S., "A Computer Program for Doing Tedious Algebra (SYMB66)", AEC Research and Development Report NYO-1480-88, New York University, New York, January 1968, 46 pages.

[10] Loev, Gerald, "SLIP List Processor", Univ. of Pennsylvania Computer Center, Philadelphia, April 1966, 57 pages.

[11] Orgass, R.J. & Waite, W.M., "A Base for a Mobile Programming System", Comm. ACM, vol. 12, no. 9, September 1969, pp. 507-510.

[12] Raphael, B. et al, "A Brief Survey of Computer Languages for Symbolic and Algebraic Manipulation", in "Symbol Manipulation Languages and Techniques", Proc. IFIP Working Conf. on Symbol Manipulation Languages, ed. D.G. Bobrow, North-Holland Pub. Co., Amsterdam 1968, pp. 1-54.

[13] Sammet, Jean E., "Survey of Formula Manipulation", Comm. ACM, vol. 9, no. 8, August 1966, pp. 555-569.

[14] Weizenbaum, J., "Symmetric List Processor", Comm. ACM, vol. 6, no. 9, September 1963, pp. 524-544.

[15] Weizenbaum, J., "Recovery of Reentrant List Structures in SLIP", Comm. ACM, vol. 12, no. 7, July 1969, pp. 370-372.

5. INDEX

3H1.* see parentheses

accuracy of numbers in expressions 2.7

ADD (add expressions - obsolete) 3.29

ADVLEL, ADVLER, ADVLNL, ADVLNR, ADVLWL, ADVLWR, ADVSEL, ADVSER, ADVSNL, ADVSNR, ADVSWL, ADVSWR (advance list reader) 3.51, 3.56

ALØAD (for use in argument list scanning) 1.24

analysis of expressions 2.35-2.40, 3.51-3.59

arithmetic on expressions 2.21-2.22, 3.29, 3.62

ASSIGN statement 1.20

attribute-value lists 3.60-3.61

BØT (bottom of list) 3.51

BREAK (decompose expression - obsolete) 3.51, 3.52

Chippewa Operating System 1.3

code generation, FORTRAN 3.18, 3.21, 3.62

CØMMØN 1.16

—— /LSQARG/ 3.21

—— /LSQCSX/ 3.6-3.7, 3.26-3.27

—— /LSQDMF/ 3.37-3.38, 3.39

—— /LSQINT/ 3.21

—— /LSQNTR/ 3.21

—— /SYMSYS/ 1.16

constraints on SYMBOLANG programs 1.15-1.16, 2.42
CØNT (contents of machine address) 1.25
Control Data 6600 1.1
core requirements see constraints
CPYTRM (copy term - obsolete) 3.1-3.2, 3.26
creation of expressions 2.1-2.3, 3.1-3.11

definition see evaluation
DELETE (delete list element) 3.26, 3.27
destruction of expressions 2.17, 3.26-3.28
determinant 2.54-2.58
differentiation 2.31-2.32, 2.45-2.46
DVSUM (divide expressions - obsolete) 3.29

entry points in SYMBOLANG 1.17-1.18
EQUAL (compare quantities) 1.25, 2.35, 2.40
6HEQUAL. 3.16, 3.40
evaluation of expressions 2.24-2.26, 2.43-2.44, 3.30-3.45
EVALUE (evaluate expression - obsolete) 3.30, 3.44
expression 1.5-1.10
    conditional —— 1.22, 2.44, 2.48
    constant —— 1.6, 2.49
    dumps of —— 2.19
    syntax of —— (input) 2.5-2.6, 3.14
    syntax of —— (internal) 1.5

factor 1.5, 3.59

GETCØE (get coefficient - obsolete) 3.51, 3.52

head of a list 1.4
HITENT (substitution in expression - obsolete) 3.30, 3.44

IBM 7094 1.3
IFTHEN (FORTRAN callable conditional expression) 1.22-1.23
3HIF. 2.44, 2.48

INHALT (contents of machine address) 1.25
INITAS (initialization routine) 1.16, 2.24, 2.42
initialization 1.16, 2.42-2.46
INITRD (reinitialize list reader) 3.51, 3.56
INLIST (input expression) 1.10, 2.5-2.11, 3.4, 3.14-3.17, 3.26, 3.40
INLSTL (insert into list) 3.1, 3.2-3.3, 3.4, 3.26
INLSTR (insert into list) 3.1, 3.3-3.4, 3.26
input of expressions see INLIST
INSBST (substitute in expression - obsolete) 3.30, 3.44
INSUBT (substitute in expression - obsolete) 3.30, 3.44-3.45
INTARG (for use in argument list scanning) 1.24
INTENT (substitute in expression - obsolete) 3.30, 3.45
INTGER (provide integer name) 1.19, 1.25
IRALST (erase list - use with caution) 3.4, 3.5, 3.26, 3.27
IRARDR (erase list reader) 3.51, 3.56
ITSVAL (obtain value of attribute) 3.60

job structure 1.15

LCNTR (level counter) 3.51, 3.57
LDIF (differentiate expression) 2.31-2.32, 2.45
level
    definition —— 3.32
    expression —— 3.30
    list —— 3.55
list 1.3-1.5
    constraints on the elements of a —— 1.5
LIST (create list) 2.1, 3.1, 3.4-3.5, 3.26, 3.27
LISTMT (test for empty list) 2.35, 2.39
LISTØN (input expression) 3.17
LØCARG (for use in argument scanning) 1.24
LØCT (test list for validity) 3.51
LØFRDR (list being scanned by list reader) 3.51, 3.57
LØØKUP (define derivative) 2.31, 2.32, 2.42, 3.4, 3.26

LPNTR (list pointer of list reader) 3.51, 3.57

LRDRCP (copy list reader) 3.51, 3.57

LRDRØV (establish list reader) 3.51, 3.56

LSQADD (add expressions) 2.3, 2.21-2.22, 3.7, 3.26

LSQCMP (compare expressions) 2.35, 2.39, 3.59

LSQCNM (compare expression symbols) 2.35, 2.39-2.40

LSQCPY (copy expression) 3.1, 3.5-3.6, 3.26

LSQCXP (copy expression) 3.1, 3.6-3.7, 3.27

LSQDEF (define value of expression symbol) 2.24, 2.25-2.26, 2.42, 3.4, 3.16-3.17, 3.26, 3.30, 3.32, 3.33-3.38, 3.43

LSQDES (erase expression) 2.17, 3.4, 3.5, 3.26, 3.38

LSQDSF (simple function definition) 3.30, 3.38-3.39

LSQGAR (get argument) 3.30, 3.38, 3.39

LSQIDR (initial derivative definitions) 2.42, 2.45-2.46

LSQINI (initial evaluation definitions) 2.24, 2.42, 2.43-2.44, 2.46, 2.47

LSQMEX (multiply expressions) 2.21-2.22, 3.7, 3.26

LSQMNL (create list) 2.1, 3.26, 3.51

LSQØUT (general output routine) 2.13, 3.19-3.22

LSQPNT (output expression) 2.13-2.15, 2.19, 3.18-3.19, 3.20, 3.21 3.22-3.24

LSQRAZ (raise expression to power) 2.21-2.22

LSQSBS (substitute in expression) 2.24, 2.26, 3.31, 3.44, 3.45

LSQTRC (truncate expression) 2.28-2.29, 3.45, 3.47-3.49, 3.52

LSQTYP (type of expression) 2.24, 2.35, 2.37-2.39, 3.53

LSQUNM (unbuild list) 1.20, 2.17, 3.26

LSQVAL (evaluate expression) 2.24-2.25, 2.37, 2.43, 3.26, 3.30 3.31, 3.32, 3.35

LSQVVL (evaluate expression) 3.30, 3.38, 3.39

LSSCPY (copy list or expression) 2.3, 3.6, 3.26, 3.27

LSTEQL (compare lists) 2.35, 2.39

LVLRV1 (raise list reader level) 3.51, 3.57

LVLRVT (raise list reader level) 3.51, 3.57

machine address 1.24-1.25

MADLFT (locate left element) 3.51, 3.54

MADNBT (locate element relative to list bottom) 3.51, 3.55
MADNTP (locate element relative to list top) 3.51, 3.55
MADØV (machine address of) 1.25
MADRGT (locate right element) 3.51, 3.55
MAKEDL (assign attribute-value list) 3.60, 3.61
MANY (add elements to list) 1.20, 2.2, 3.7
MØNØ (provide integer name) 1.20
MTDLST (remove attribute-values) 3.60, 3.61
MTLIST (empty list) 3.26, 3.27, 3.28

NAMEDL (find attribute-value list) 3.60, 3.61
NAMTST (test for being a list) 3.51, 3.52
NEWBØT (add to bottom of list) 3.1, 3.7, 3.8
NEWTØP (add to top of list) 3.1, 3.8
NEWVAL (new value for attribute) 3.60-3.61
notation 1.3-1.11
NULSTL (break off elements as a list) 3.1, 3.8, 3.26
NULSTR (break off elements as a list) 3.1, 3.9, 3.26
NUMPY (multiply expression by number - obsolete) 3.29
NXTLFT (insert element to left) 3.1, 3.9
NXTRGT (insert element to right) 3.1, 3.10

operators, hierarchy of 2.8
ordering 1.10, 1.12-1.14, 2.39-2.40, 3.59
output of expressions 2.13-2.15, 2.19-2.20, 3.18-3.24
overwritng, execution time 1.22
ØVWRT (execution time overwriting utility) 1.22

parentheses (represented by 3H1.*) 1.7, 2.8, 2.43, 2.45, 2.47-2.48
    removal of —— 2.49-2.54
7HPARTIAL 2.31
PØPBØT (pop off bottom element) 3.26, 3.27
PØPMID (remove an element) 3.26, 3.27-3.28
PØPTØP (pop off top element) 3.26, 3.28
power 1.5
PØWER (extract power from expression) 3.51, 3.52

preprocessor 1.2, 3.62-3.73
primitives, SLIP 1.25-1.26
PRIPUT (output expression - obsolete) 3.24
PRLSTS (dump list or expression) 2.19
PUTLST (build expression - obsolete) 3.1, 3.10

QUEST, preprocessor 1.2
2HQ. 2.44

RCELL (return cell) 3.26, 3.28
reader, list 3.55-3.58
reader, sequence 2.35-2.37, 3.5-3.6, 3.53-3.54
REAL (provide floating point name) 1.20
recursion 1.20-1.21
REED (read reader) 3.51, 3.58
RUN 2.3 (FORTRAN compiler) 1.1, 1.15-1.18

SAME (provide floating point name) 1.20
SBARGS (count arguments in call) 1.23-1.24
SBST (substitute in expression - obsolete) 3.30, 3.45
SCOPE 3.1.2 operating system 1.1, 1.15-1.18
SELECT (satisfy externals) 1.15
SEQLL (advance sequence reader to left) 2.35-2.37, 3.53, 3.54
SEQLR (advance sequence reader to right) 2.35-2.37, 3.53, 3.54
SEQRDR (sequence reader) 2.35-2.37
SEQSL (advance seq. reader structurally left) 3.51, 3.53, 3.54
SEQSR (advance seq. reader structurally right) 3.51, 3.54
simplification see ordering, SMPL
SLIP list processing package 1.1, 1.3-1.5, 1.18
SLIP program 1.15
SMPL (simplify expression) 1.12
SØLVE (solve linear equation) 3.51, 3.52
STRDIR (store directly) 1.19, 1.22, 1.25
STRIND (store indirectly) 1.25
SUB (subtract expressions - obsolete) 3.29
SUBSBT (substitute at list bottom) 3.1, 3.10-3.11

SUBST (substitute in list) 3.1, 3.11
substitution see evaluation
SUBSTP (substitute at list top) 3.1, 3.11
SUBT (substitute in expression - obsolete) 3.30, 3.45
SUMPY (multiply expressions - obsolete) 3.29
SYMB66 ii

term 1.5, 3.59
TERM (terminate VISIT) 1.20, 3.22, 3.38
tests see analysis
THENIF see IFTHEN
TØP (top of list) 3.51
TRCAL (truncate expression - obsolete) 3.51, 3.53
TRUNC (truncate expression - obsolete) 3.50
truncation 2.28-2.29, 3.47-3.50
TSTCØN (test if expression is constant - obsolete) 3.51, 3.53

unit, input/output 1.16, 2.13

VALARG (for use in scanning argument lists) 1.24
VISIT (recursive use of FORTRAN code) 1.20, 3.22, 3.36, 3.38, 3.39, 3.42
2HV. 2.44

This report was prepared as an account of Government sponsored work. Neither the United States, nor the Commission, nor any person acting on behalf of the Commission:

A. Makes any warranty or representation, express or implied, with respect to the accuracy, completeness, or usefulness of the information contained in this report, or that the use of any information, apparatus, method, or process disclosed in this report may not infringe privately owned rights; or

B. Assumes any liabilities with respect to the use of, or for damages resulting from the use of any information, apparatus, method, or process disclosed in this report.

As used in the above, "person acting on behalf of the Commission" includes any employee or contractor of the Commission, or employee of such contractor, to the extent that such employee or contractor of the Commission, or employee of such contractor prepares, disseminates, or provides access to, any information pursuant to his employment or contract with the Commission, or his employment with such contractor.